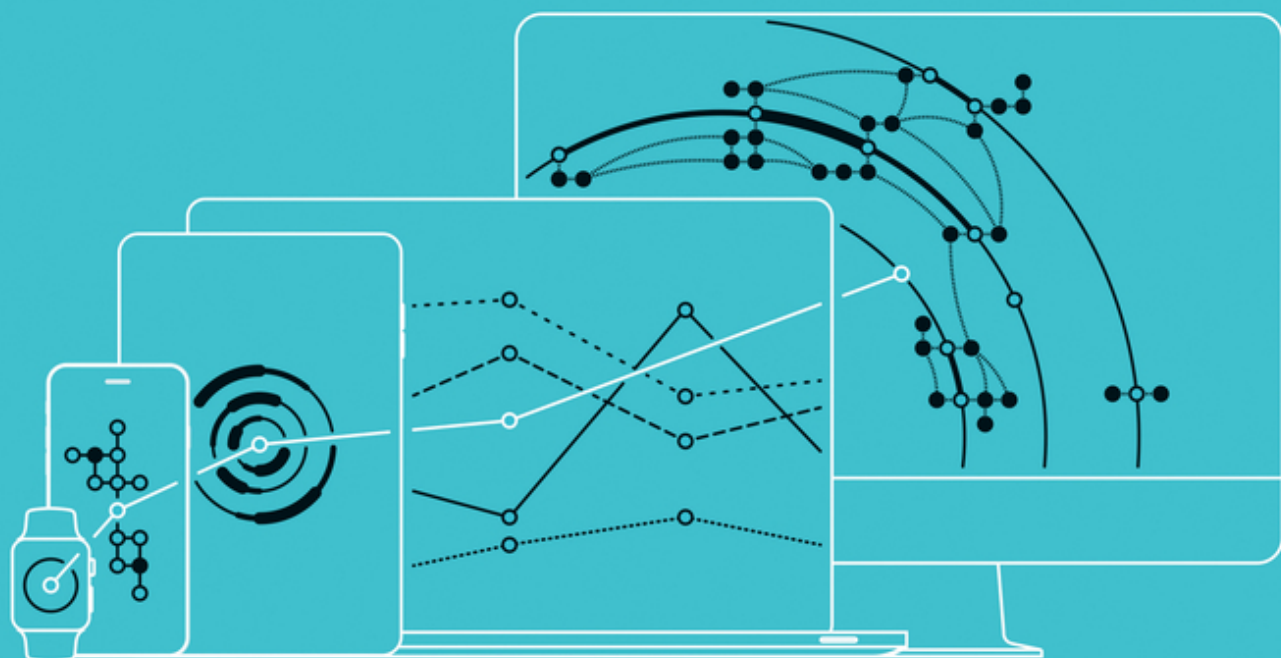


Bill Hinderman

Building Responsive Data Visualization for the Web



WILEY

Building Responsive Data Visualization for the Web

Building Responsive Data Visualization for the Web

Bill Hinderman

WILEY

Building Responsive Data Visualization for the Web

Published by
John Wiley & Sons, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2016 by John Wiley & Sons, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-1-119-06714-6
ISBN: 978-1-119-06713-9 (ebk)
ISBN: 978-1-119-06720-7 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permissions>.

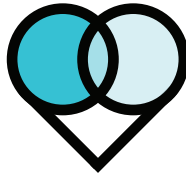
Limit of Liability/Disclaimer of Warranty: The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or website may provide or recommendations it may make. Further, readers should be aware that Internet websites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services please contact our Customer Care Department within the United States at (877) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at <http://booksupport.wiley.com>. For more information about Wiley products, visit www.wiley.com.

Library of Congress Control Number: 2015951125

Trademarks: Wiley and the Wiley logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.



For my grandmother, Mary.

You showed me that the Internet is for people.

About the Author



Bill Hinderman is a software engineer and designer from Chicago. He is the lead site optimization UI engineer at Orbitz Worldwide (orbitz.com), and a space cadet and designer at Starbase Go (starbasego.com). He designs, prototypes, develops, and A/B tests experimental new products for customer-facing brands.

He works alongside artistic and entrepreneurial individuals to craft innovative web experiences. Bill received his B.S. in computer science with a specialization in UI development from the University of Illinois in 2012. He's a speaker at conferences on the future of web development, data visualization, and cross-platform design.

Bill is an avid runner and cocktail-maker, and he can wear the hell out of a suit. When he was four, he penned *My Book About Me*, in which he stated that he was glad he wasn't a giraffe, and that his least favorite thing to do was sitting. Both of those statements hold true to this day.

A

About the Technical Editor

Randy Krum is an infographics and data visualization designer, author of the book *Cool Infographics: Effective Communication with Data Visualization and Design* (Wiley, 2014), and instructor of Infographics and Data Visualization Design at Southern Methodist University's Continuing and Professional Education program (CAPE). Randy also runs the popular website Coolinfographics.com. Started in 2007, the site has grown into one of the most influential infographic sites online, handling up to 50,000 unique visitors per day. He is the founder and president of InfoNewt, a design company that creates infographics and visualizations for clients used for both online marketing and internal communications. Randy speaks at conferences, universities, corporate events, and government agencies about infographic design, data visualization methods, visual content marketing, and the effective use of visual information. Learn more at RandyKrum.com.

Credits

Acquisitions Editor

Jim Minatel

Project Editor

Adaobi Obi Tulton

Technical Editor

Randy Krum

Production Editor

Rebecca Anderson

Copy Editor

Luann Rouff

Manager of Content Development and Assembly

Mary Beth Wakefield

Marketing Director

David Mayhew

Marketing Manager

Carrie Sherrill

Professional Technology & Strategy Director

Barry Pruett

Business Manager

Amy Knies

Project Coordinator, Cover

Brent Savage

Interior Designer and Composer

Happenstance Type-O-Rama

Proofreader

Nancy Carrasco

Indexer

Johnna VanHoose Dinse

Cover Designer

Bill Hinderman

Cover Image

Bill Hinderman

Acknowledgments

I want to thank Carol Long from John Wiley & Sons. After opening for the morning coffee break at the Data Visualization Summit in Boston, you walked up to me with a coffee in one hand and a business card in the other, and asked if I would like to write a book. Without your willingness to approach some ridiculous 24-year-old kid, this whole thing never would have happened.

I also have to give my most heartfelt appreciation to my project editor, Adaobi Obi Tulton. You were equal parts patient and stern regarding deadlines and deliverables, in a way that made this long book-making process enjoyable.

I also owe a massive thanks to Randy Krum for not only acting as technical editor on the book, but also giving me advice—as someone who has done this same thing before—on pitfalls to avoid, and paths to take.

Finally, I want to offer apologies to any family members, friends, coworkers, and strangers I was rude to, short with, or otherwise unpleasant to be around during deadline weeks. I promise it wasn't you.

Contents

Introduction xxv

PART ONE: Creating the Responsive Web

The Mobile Web 3

<i>How We Got Here</i>	4
<i>The Web's History</i>	5
<i>The Rise of Mobile Devices</i>	8
<i>The Mobile Tipping Point</i>	10
<i>The Mobile Web Today</i>	13
<i>Mobile Web Considerations</i>	13
<i>Benefits of the Mobile Web</i>	15
<i>Building for the Most Users</i>	17
<i>Mobile Web Design</i>	17
<i>The Way To Build Today</i>	18
<i>A Taste of Progressive Enhancement</i>	20
<i>Summary</i>	23

02 Responsive Web Design Tenets 25

<i>The Gist</i>	26
<i>Building for the Limited User</i>	27
<i>Building for Every User</i>	28
<i>Adaptive vs. Responsive</i>	29
<i>The Broad Adaptive</i>	29
<i>Limiting Your Adaptive Design Definition</i>	33
<i>Desktop-First vs. Mobile-First</i>	36
<i>Four Principles</i>	38
<i>Be Universal</i>	39
<i>Be Flexible</i>	43
<i>Be Economical</i>	46
<i>Be Specific</i>	47
<i>Seven Points of Focus</i>	49
<i>Measure</i>	50
<i>Flow</i>	51
<i>Nesting</i>	54
<i>Endpoints</i>	57
<i>Breakpoints</i>	59
<i>Typography</i>	63
<i>Images</i>	66
<i>Summary</i>	69

03 Working with a Flexible Grid 71

<i>The Gist</i>	72
<i>What Is a Grid System?</i>	72
<i>Flexible Units</i>	75
<i>The Meticulous Client</i>	75
<i>Moving to Relative Units</i>	79

<i>Using a Grid System</i>	81
<i>Another Project</i>	82
<i>Branching Out</i>	93
<i>Creating a Reusable, Flexible Grid (in Five Easy Steps)</i>	93
<i>Unit</i>	95
<i>Gutter</i>	95
<i>X-Count</i>	95
<i>Columns</i>	96
<i>Rows</i>	96
<i>Putting It All Together</i>	97
<i>Summary</i>	98
<i>Try It</i>	98

04 Designing Responsive Layouts with CSS **101**

<i>The Gist</i>	102
<i>A Brief History of CSS</i>	102
<i>You've Come a Long Way, Baby</i>	103
<i>Responsive Layout Design</i>	118
<i>The Media Query</i>	118
<i>Starting from the Bottom</i>	127
<i>The Reusable Responsive Grid</i>	144
<i>Building a Reusable Grid</i>	145
<i>Building a Responsive Grid</i>	147
<i>Summary</i>	148
<i>Try It</i>	149

05 Enhancing with JavaScript 151

<i>Using JavaScript</i>	152
<i>JavaScript Enables Rich Interactivity</i>	153
<i>Remember Progressive Enhancement</i>	155
<i>Sensible Defaults</i>	157
<i>Responsive JavaScript</i>	163
<i>Server-Side Feature Detection</i>	164
<i>Building from CSS</i>	164
<i>JavaScript's matchMedia Object</i>	166
<i>Limber Up</i>	168
<i>AJAX</i>	168
<i>Perceived Performance</i>	172
<i>Summary</i>	182
<i>Try It</i>	182

PART TWO: Creating Responsive Data Visualization

06 Data Design: An Abridged History 187

<i>Learning From Data</i>	188
<i>Data, Information, and Knowledge</i>	189
<i>Size Matters</i>	191
<i>Data's Data</i>	192
<i>The Big Pile</i>	193
<i>Extracting Meaning</i>	193
<i>Communicating Meaning</i>	195

<i>What You Get from the Web</i>	210
<i>Interactivity</i>	211
<i>A Taste of Responsive</i>	212
<i>Summary</i>	214

07 Responsive Data Visualization Tenets 217

<i>Designing Content-First</i>	218
<i>Still Just the Web</i>	219
<i>Distilling Data</i>	220
<i>Revisiting Responsive Design Principles</i>	229
<i>Be Universal</i>	230
<i>Be Flexible</i>	232
<i>Be Economical</i>	235
<i>Be Specific</i>	240
<i>Seven Points of Focus</i>	241
<i>Measure</i>	242
<i>Flow</i>	245
<i>End points</i>	247
<i>Breakpoints</i>	249
<i>Grouping Points</i>	252
<i>Images</i>	253
<i>Layering</i>	256
<i>Responding to Data</i>	258
<i>Know Your Audience</i>	259
<i>Know Your Data</i>	260
<i>Summary</i>	261

08 Thinking Small 263

<i>Designing for the Smallest Canvas: No Canvas</i>	264
<i>Building for a Blind User</i>	266
<i>The Good API</i>	269
<i>The API-First Team</i>	278
<i>The Tiny Canvas</i>	279
<i>The Fitness App</i>	279
<i>Layering</i>	283
<i>Enhancing Efficiently</i>	285
<i>Growing Up</i>	286
<i>First, Change Everything</i>	287
<i>They're Just Numbers</i>	290
<i>Summary</i>	291

09 Asset Dependence 293

<i>Dynamic Data</i>	294
<i>Dipping Your Toe In</i>	294
<i>Data versus Screen</i>	298
<i>Reacting to Data Changes</i>	301
<i>Tying Visualization to Screens</i>	308
<i>Grouping Again</i>	309
<i>Summary</i>	312
<i>Try It</i>	313

10

Code-Driven Visualization 315

<i>Unknown Inputs and Outputs</i>	316
<i>To the User</i>	317
<i>To the Data</i>	317
<i>Putting It Together</i>	320
<i>Using D3.js</i>	320
<i>Getting Started with D3.js</i>	321
<i>Data and the DOM</i>	348
<i>Building Responsive Data Visualization for the Web</i>	354
<i>Client Responsiveness</i>	355
<i>Data Responsiveness</i>	361
<i>The Bigger Picture</i>	363
<i>Rational Defaults</i>	363
<i>Driving Design with Data</i>	371
<i>Clear, Not Clever</i>	372
<i>Summary</i>	372
<i>Try It</i>	373

11

Building the Future-Friendly Web 375

<i>The Future of Data Design</i>	376
<i>Growing Stockpiles of Data</i>	377
<i>Setting a Standard for Data Markup</i>	378

<i>Embracing Uncertainty</i>	379
<i>Where Complexity Lies Today</i>	380
<i>The Future-Friendly Manifesto</i>	381
<i>Responsible Web Design</i>	385
<i>Building API-first</i>	386
<i>Respecting Progressive Enhancement</i>	386
<i>Innovating</i>	388
<i>Standardizing</i>	388
<i>Summary</i>	389

PART THREE: Additional Resources

Resources 393

<i>Responsive Data Visualization</i>	394
<i>Grids</i>	394
<i>Infographic Design</i>	394
<i>Responsive Design</i>	395
<i>D3.js</i>	395
<i>More Resources Online</i>	396

Index 397

00

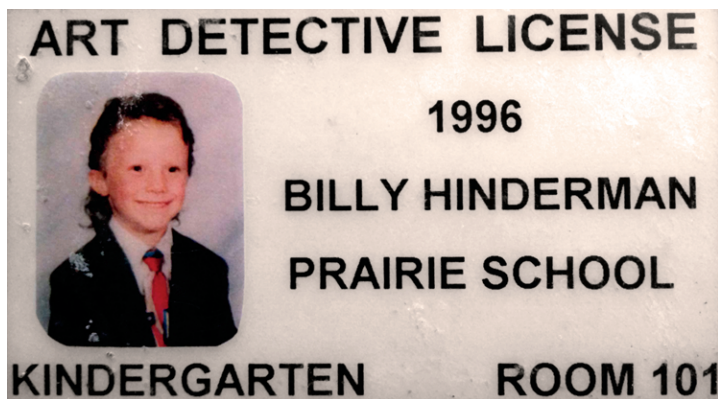
Introduction

Before we do anything, I want to tell you about my first experience with the web.

It was 1996, and I was at home, sitting at my mother's Windows 3.1 machine. We had recently heard about this "Internet" from my grandmother, who had been using it to communicate with a friend living in Hong Kong.

*To us, this was science fiction. I was six years old, however, and science fiction wasn't as sharply differentiated from science fact as it (sometimes) is today, so it never occurred to me that this **shouldn't** be possible.*

This was my first time at the computer in a week, not because I wasn't using a computer regularly already, but because a week prior I had gotten up very early, logged onto my mother's computer, and, having memorized the keystrokes and icons, rather than the actual words, navigated to the Control Panel and turned every single background, foreground, and text color white. When I got home from kindergarten that day, I was greeted less kindly than with the typical chocolate milk and *Happy Days* reruns.



Someone who can't differentiate between science fiction and science, fact

Therefore, a week after dismantling my minimalist design, I was about to experience the Internet. My next-door neighbor at the time was also named Billy; and in a grand act of vanity, I decided I would send my first e-mail message to the only other person I knew who shared my name.

I opened the e-mail client and slowly chicken-pecked the following:

Isn't technology great?!

-Billy H.

In little-kid time, it took somewhere between one second and 700 years for this e-mail to be sent. Immediately, I called the other Billy on the phone to see if he had

received it. He had to get off the phone to check. Eventually, he responded with a poignant piece of writing that I carry with me to this day:

yes!!

-Billy R.

The e-mail client, sitting in its own white window on a background I had begrudgingly recolored a neon yellow, wasn't much more than a WYSIWYG editor and a list of contacts—the same as it looks today.

At six, I felt like I had just sent a message to Hong Kong.

Five years later, I was sitting at my grandmother's house. My great-grandmother, who was a rock star and could beat any of you at Scrabble so don't even try, was in a nursing home that housed both a group of eccentric old retirees and a surprisingly tech-savvy administrator.

I had decided, after losing at Scrabble, that I would make my first website: a face book for all the people on my great-grandmother's floor of the nursing home, a way to let them say hello to their loved ones online. The administrator was over the moon about this. At eleven, I had to open an MS-DOS prompt and manually connect to the free web hosting included with my mother's e-mail address at the time, then FTP individual HTML files to the server. That website eventually led to my winning a Jefferson Award for Public Service and renting a (small) tuxedo to talk about the Internet, surrounded by adults who were making a tangible difference in the world.

I was once again living in the world of science fiction.

Five years later, I had my first cellphone. It did not have any Internet-connected features. At the time, having Internet connectivity on your phone meant one of two things:

- You were using it for e-mail.
- You were in a commercial for cellphones.

I had no inclination that the little clamshell in my pocket would morph into the primary way that I now access the web every day.

A year later, the iPhone would launch.

Today, we hold our Internet to a higher standard. We (I, because I know who I am and what I do for a living, and you, because you have opened this book, which means I can make wild assumptions about you) make things for the web. We make websites, and web apps, and landing pages, and blogs, and magazines, and dashboards, and we do all of this because we have a vested interest in making the Internet something amazing.

My degree is in computer science, with a focus in UI engineering and human-computer interaction. I shimmied backward into web design because I missed changing all of my mom's system colors to white, and because nobody else I was working with ever wanted to also be the designer when we were up at 4:00 a.m. regurgitating data structures.

I cut my teeth on data visualization toward the end of my college career. A friend (with whom I would go on to form my studio) and I were working on a comically uncreative idea: tracking the level of crowd, drink and food specials, and how many friends were at any given venue on a given night. (It was far from the next Facebook, but we were young and shiny and dumb.) Never before, and often since, I felt the intense joy of watching a complex set of numbers turn into something readable, approachable, even friendly.

Because I've lived in this dual role of developer and designer for the entirety of my professional career, I have never been able to separate the visual design of how a website, a visualization, a *pixel* might look from its actual implementation—and I think that this is an important way to look at what you create for the web.

Our industry is responsible for some of the most amazing advances in the past 25 years; but it's also responsible for the tidal wave of absolute garbage being shoved into consumers' faces on a daily basis. Not only do we create content, we also create the vehicles by which other people create *and consume* content.

But I don't want to scare you.

We're the good guys. We're designers. We get to decide what is made, how it works, and exactly how consumers interact with it; but there's one thing that has always left me speechless: We've made it this far without a plan.

The Internet looks and acts the way it does simply because we're all rowing in the same direction. There is no one place to get this year's model of the Internet. There are no vintage web shows that sweep through the Midwest during the summer; and there is no one company, group, team, or person deciding the direction of the Internet. If an *Internet thing* is made well, defended adequately, and presented broadly, it becomes the standard.

It's the world's most successful design by consensus.

Who You Are (Now)

So you're reading this book. Cool. Thanks. I'd love to know what you think of it. It's absolutely *not* for everyone. You're here because like me you are a maker of web things,

...and some of those web things happen to be data visualizations.

...and all of those web things suddenly have to work on a four-inch-by-six-inch screen.

...and you're now somewhere between "I'm feeling OK about this" and "I didn't know my palms could even get this sweaty."

We have our work cut out for us.

From my experience working with massive piles of data at Orbitz Worldwide (orbitz.com), to visualizing a few simple ingredients in cocktail recipes at The Cocktail Gui.de (thecocktailgui.de), I know that the trick to getting good at this sort of work is to do a lot of it. In the case of the latter, we always knew what we wanted to fix. Critique was easy. It was this work, this process, this heartbreak of eating our firstborn that actually led us to *fixing* it.

Visualization design, just like web design, takes a whole lot of time to get right.

In my case, it typically takes walking down two or three *completely wrong* paths before I realize the answer I had been looking for. Our job is to take large data

sets and make them simple, digestible, and persuasive through translation into the visual space. That's a tall order, and one that can't be filled in a fast-food restaurant sort of way.

This book will help you on your path to becoming a competent designer of responsive data visualizations.

If you plan to sit down at a coffee shop and breeze through the next 300 pages, glancing occasionally at the examples, you will probably learn *a few* things. You might even learn *more than a few* things; but to truly get the worth from these lessons, you need to take your time, follow along with the practice problems, and move around the book as you focus on one problem or another.

What You'll Learn in Class Today

This book is about one thing: responsive design's relation to data visualization.

Over the years, in more and more projects, I was asked by business stakeholders or product managers or clients to create “a mobile version” or “a tablet version” of something that a designer already thought through very carefully in the context of a large desktop screen with a keyboard and mouse. Then the request changed to “make this responsive,” while being handed the same desktop-context *picture of a website*. This idea is covered heavily in the first few chapters, but the point is that responsive design has moved from revolution to buzzword.

The thing I like about buzzwords is that they can sometimes still be meaningful concepts. For example:

- Independence
- Buy-one-get-one
- Social media
- Responsive design

Obviously that last one is where we'll be spending the bulk of our time. If you don't know anything about responsive design, that's totally fine; the following chapters will walk you through everything— basic, intermediate, and even some advanced.

Once we're all on an even playing field, we're going to focus on a concept that was somehow overlooked when responsive design was being architected: data visualizations are just content. Fluid grids, CSS media queries, and progressive enhancement are the technical cornerstones of responsive web design. Moreover, as you'll be able to tell everyone you know, these same three techniques can be applied to data visualizations to *make* them respond to your users.

You'll learn design techniques; you'll learn the code to back those techniques; and you'll learn why you should bother even thinking about this to begin with.

This book outlines a way to move toward one ubiquitous, ergonomic web, regardless of device. As a designer/developer, you're solving for 100% parity not only in layout, text, and images, but also in rich, interactive content.

How This Book Works

So here's the plan: You have a book in front of you. Hooray. These days, however, we live in a futuristic, science-fiction world, and books aren't *just* books anymore, so you also have the following:

- A reference of responsive web design techniques that you can look up on a whim
- A wealth of annotated examples to explore and blatantly rip off
- A series of guided practice exercises to learn the basics and the not-so-basics of responsive web and visualization design
- An online resource that works alongside this more-than-a-book to help you get the most out of your learning experience

Since we're building for web technologies, and it's very hard to get the web on paper, most of the figures and visualizations in this book are annotated, and can be viewed on responsivedatavisualization.com. All the practice material and examples can be found there as well, along with a list of related articles and news, which I will continue to update long after you've finished reading this book and added it to that shelf of items you keep in case the creative director walks by.

How This Book Is Organized

This book is separated into two different parts, each with its own focus. The parts have been written to be entirely self-contained, so you don't need to read the book from cover to cover if you are already familiar with certain subjects.

The sections are separated as follows:

- Part I, “Creating the Responsive Web,” begins with a brief history of how the mobile web came into being. It establishes a definition for the mobile web and describes how we, as web designers, have begun to interact with and develop for it using responsive web design techniques. It then outlines those techniques that serve as the cornerstones of responsive web design.
- Part II, “Creating Responsive Data Visualization,” begins by walking through the science of data visualization on the web. It then takes the lessons from responsive web design and combines them with techniques used to create beautiful web visualizations.

Who You Are (300 Pages from Now)

After you have finished reading this book, you'll be a conduit for moving the Internet toward a more perfect home for data. If you follow along (or even if you just copy and paste), you'll have the necessary toolset to build rich, interactive data visualizations that fit comfortably on any screen type.

It's my hope that you'll use your knowledge to approach each new project with a flexibility and adaptability that enables you to solve for emerging screen types and interactions without starting from scratch every time.

Future-Friendly Feedback

Clearly, no book has all the answers. Responsive data visualization is as much about a way of thinking as it is about a technical approach to a design problem. There are plenty of you reading this who work at the leading edge of this creative wave, and I encourage you to add your thoughts to the conversation:

Building Responsive Data Visualization for the Web:

responsivedatavisualization.com

Twitter: [@billHinderman](https://twitter.com/billHinderman)

Facebook: facebook.com/billHinderman

LinkedIn: linkedin.com/in/williamhinderman

Me: bill@billhinderman.com

Starbase Go: starbasego.com

Why This Matters

In the next few hundred pages, you will read about the overwhelming shift toward mobile in the Internet landscape. The most important insight I can share about this is that every time we have made the assumption that someone *won't* do something on mobile, we've been proven wrong.

There's no reason to think that data science is a unique case.

However, that's only half of the conversation. Responsive web design, and responsive data visualization are not simply a way to make our content accessible on smaller screens. We need to build an ergonomic web that feels natural regardless of device type. Whether it's a desktop, a phone, a television, or some type of screen that hasn't been invented yet, you want to design in such a way that your content is always accessible and understandable to those looking at it.

Let's get started.

Building Responsive Data Visualization for the Web

ONE PART

Creating the Responsive Web

CHAPTER 01: [The Mobile Web](#)

CHAPTER 02: [Responsive Web Design Tenets](#)

CHAPTER 03: [Working with a Flexible Grid](#)

CHAPTER 04: [Designing with CSS](#)

CHAPTER 05: [Enhancing with JavaScript](#)

Or

01

The Mobile Web

Before discussing responsive data design, it is important to set the stage with a brief overview of how we got here. The evolution of the mobile web does not exist in a vacuum; it is built upon the foundation of 25 years of the Internet's growth.

In this chapter, we'll touch upon the beginning of the consumer Internet, and how it became commercialized. We will then move on to the shift from desktop computers to mobile devices as the primary way to access the web.

Finally, you will gain an understanding for what the mobile web's landscape looks like today by exploring how the screens currently used came to be, and what will become their logical successors.

Plus, you will end up with great conversation starters for the world's nerdiest cocktail party.

How We Got Here

Think about the device you have in your pocket or in your hand right now. You have an absolutely godlike amount of computing power inside a device the size, shape, and fragility of a toaster pastry. Not only can it do miraculous things like calculate your age from a picture of your belly button, but it can also talk to all of the other devices on the planet and shame you for missing your morning run.

Back in the 1950s, though, computers were massive room-filling calculators that couldn't say anything to one another. Computer science labs across the United States, Great Britain, and France were working on that "couldn't say anything to one another" part, though. Throughout that decade, and into the 1960s, packet networking systems (what humans call the Internet) had been contracted by the United States Department of Defense. The first ARPANET (what would become the first network to use the Internet Protocol) link was established between the University of California, Los Angeles (UCLA) and the Stanford Research Institute on October 29, 1969. The messages went as follows:

"We set up a telephone connection between us and the guys at SRI ...," Kleinrock ... said in an interview: "We typed the L and we asked on the phone, 'Do you see the L?'"

"Yes, we see the L," came the response.

We typed the O, and we asked, "Do you see the O."

"Yes, we see the O." Then we typed the G, and the system crashed.

FROM GREGORY GROMOV'S ROADS AND CROSSROADS OF THE
INTERNET HISTORY (WWW.NETVALLEY.COM/CGI-BIN/INTVAL/NET_HISTORY.PL)

Access to the ARPANET was expanded in 1981 with the creation of the Computer Science Network. A year later, the Internet protocol suite (TCP/IP) became the standard networking protocol on that network.

The Web's History

So, the history of the Internet begins in the late 1950s when those big computers couldn't talk, and it takes 19 years for the first conversation. So, naturally, let's begin our lesson in 1980, with the history of the web.

The web is a way of accessing information *over* the Internet. That is, it's an information-sharing model that is built on top of the Internet, using a standard hypertext transfer protocol (HTTP).

Tim Berners-Lee was an English-born contractor working at the European Organization for Nuclear Research (CERN) in Switzerland. In 1980, he became fascinated with the concept of hypertext. In the simplest terms, hypertext is text displayed in a digital form that references and provides linked access to other text. Berners-Lee built ENQUIRE, a hypertext database system to catalog people and software models. If you've ever used a wiki, like Wikipedia, you can grasp the core of what he was building with ENQUIRE. It was a database of hypertext information within which other related information was accessible through hyperlinks.

Early Days

By the mid-eighties, Berners-Lee had left and returned to CERN, still working on a solution to a seemingly intractable problem: Physicists around the world lacked a way to communicate data uniformly. In 1984, TCP/IP protocols were installed on a few machines at CERN, making it, at the time, the largest collection of *things that make the Internet the Internet* in all of Europe.

In 1989, Berners-Lee wrote a proposal for a "large hypertext database with typed links." He began creating this database on a NeXT computer workstation, and called the project the World Wide Web.

By the end of 1990, the World Wide Web was beginning to take shape as something similar to what we know today. Hypertext Transfer Protocol (HTTP) and Hypertext Markup Language (HTML) were in a working state, and the first HTTP server software and web server went online. Accompanying these were the first web pages, which described the World Wide Web project itself. Figure 1-1 shows the earliest known website. The first web browser also came online, as a project housed at CERN.

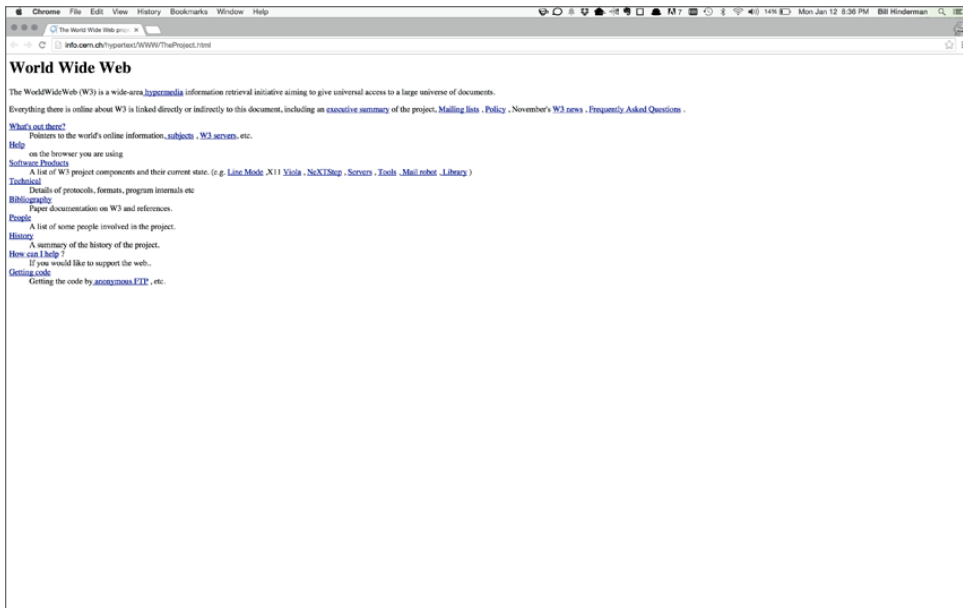


FIGURE 1-1

The first website, now housed at <http://info.cern.ch/hypertext/WWW/TheProject.html>

A pivotal moment came on August 6, 1991. Berners-Lee posted a short summary of what was then called the “World Wide Web” project on the alt.hypertext newsgroup. The summary’s opening was deceptively simple, considering its significance:

The WorldWideWeb (WWW) project aims to allow links to be made to any information anywhere.

TIM BERNERS-LEE

This summary marked the web’s introduction as a publicly available Internet service, rather than just a private project at CERN.

The Consumer Internet

The web was first made publically popular by the Mosaic browser, which was launched by a team at the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign (in a building where

I once spent eleven hours as an undergrad trying to get an MS Paint drawing of a mouse to solve a maze programmatically).

The browser was graphical (as opposed to strictly textual as in some previous browsers), and became popular due to its rapid feature growth and support of multimedia.

After graduating from the University of Illinois, the team's lead, Mark Andreessen, formed Mosaic Communications Corporation to develop the browser in a commercial capacity. In April 1994, the company changed its name to Netscape, and the browser was renamed Netscape Navigator.

That same year, PC sales reached a respectable 59 million (M) units shipped.

The Rise (and Fall) of PCs

By 1996, that number had jumped to nearly 70M units sold per year—a trend that only escalated, reaching close to 150M PCs sold in 2000. And during this period, the web continued to expand and occupy a greater part of the digital landscape.

An entire industry grew out of the seemingly limitless potential of the web. The “dot-com bubble” refers to this period, from roughly 1997 through 2000, during which startup companies with entirely Internet-based businesses were formed and received venture capital, intending to change the world by ushering in a new renaissance—one in which we would all have golden toilets and hovercraft and a soda fountain in our kitchen...

Oh.

So it wasn't limitless, but the companies that *could* survive did; and despite a one-year slowdown, PC sales continued to grow. Along with them, new browsers entered the market and gained popularity. Microsoft launched Internet Explorer, Mac had Safari, along with open-source projects such as Firefox and Opera. More recently Google Chrome has taken the lion's share of the market. Figure 1-2 shows the amount of PCs sold annually from 1995 through 2007.

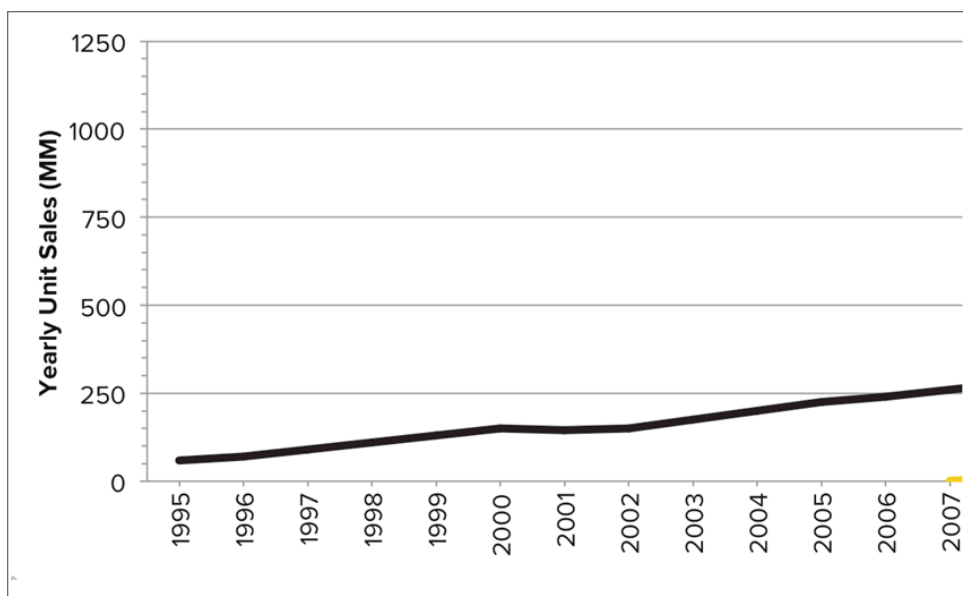


FIGURE 1-2

PC sales from 1995–2007

The Rise of Mobile Devices

During this time, it turns out we weren't all in such awe of PCs and the Internet that we decided to stop making other technological advances. Cell phones were evolving from simple green-screened boxes, to sophisticated full-color screens used to play Snake, to machines that began to reach the web.

In the meantime, the web itself continued to grow and evolve. The basic timeline of web markup advances can be simplified to the graph shown in Figure 1-3.

You may have noticed that Figure 1-2 ends at 2007. Of course, the Internet didn't end that year. The diagram stops there because something transformational emerged that year: the iPhone (see Figure 1-4).

From this point forward, mobile phones began to look and act much smarter, hence their new name “smartphone.” With these little pieces of glass, we could now interact with not just a subset of the web, or some web-like features, but the entire web.

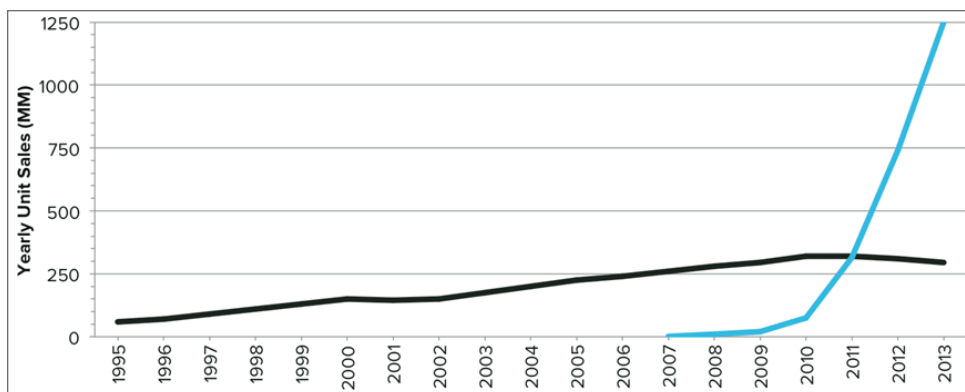


FIGURE 1-5

PC sales and mobile device sales from 1995–2013

As you can see, mobile device sales beginning in 2007 had a few good years, and then several great years. We're still in that *great* category. Mobile device sales have skyrocketed and, as of yet, show no sign of slowing.

The Mobile Tipping Point

In 2011, mobile device sales actually surpassed PC sales (see Figure 1-6). A market already saturated with smartphones received a further sales acceleration with the introduction of a third class of devices: tablets.

According to a Pew Internet survey (www.pewinternet.org/2012/06/26/cell-internet-use-2012/), 55% of Americans said they had used a mobile device to access the Internet in 2012. More shocking than that, 31% of these mobile users said that those devices are the *primary* way in which they access the web.

Mobile-Heavy Usage

An absolutely ludicrous misconception (that I hear nearly daily) about mobile devices is that it is totally fine for them to offer only a subset of the desktop version's content. Misinformed product managers make the argument that users only need quick, focused tools on their phones. To determine whether this reasoning has any merit, we can look at numbers from some of the top-visited websites in the world.



FIGURE 1-6

In 2011, yearly mobile device sales outpaced PC sales.

In 2009, less than two years after the mobile tipping point, Facebook’s mobile users accounted for roughly 13% of its total traffic (20M of 150M). Even in that short time frame, that is not a negligible number. However, by mid-2012, mobile users accounted for more than 50% of Facebook’s traffic. As of summer 2015, a staggering 78% of monthly U.S. Facebook traffic comes from mobile devices (newsroom.fb.com/company-info/#statistics). The other social media giant, Twitter, asserts that 80% of its entire user base is primarily using mobile devices (<https://about.twitter.com/company>).

Outside of social networking, more than half of all YouTube views now come from mobile devices (www.youtube.com/yt/press/statistics.html). In 2015, year over year mobile revenue on YouTube is up 100%.

Pew Research has reported that 50% of teenage (13–17) smartphone owners use the Internet mostly or exclusively on that device (www.pewinternet.org/2015/04/09/teens-social-media-technology-2015/). Similarly, 45% of adults between the ages of 18–29 mostly use the Internet on their mobile devices.

Mobile Commerce

“But wait,” you may be thinking, “those are only multi-billion-dollar social networks that changed the way we interact on a daily basis. What about e-commerce?” Here’s the thing:

In 2011, \$4 billion of eBay’s gross sales came from mobile shoppers. In 2012, this number grew to \$13B. In a rather ballsy fashion, eBay expected this trend to hold, and projected that 2013 sales from mobile would reach \$20B (techcrunch.com/2013/01/16/ebay-and-paypal-expect-to-do-20-billion-each-in-2013-mobile-commerce/, techcrunch.com/2011/10/09/ebay-vp-steve-yankovich-en-route-to-4b-in-gross-mobile-sales-tctv/). They were pleasantly surprised by an extra \$2B in mobile sales on top of that.

Moreover, not only were sales from mobile browsing astronomical, 36 million of eBay’s incremental accounts and users registered on mobile, accounting for 40% of new users (www.mobilecommercedaily.com/ebay-downplays-offline-retail-play-as-mobile-volume-swells). In 2013, I couldn’t name a single person who hadn’t either

- had an eBay account for at least five years, or
- decided they were never going to need one.

A property then owned by eBay, PayPal, experienced a similar surprise. They, too, projected mobile sales volume to reach the \$20B mark in 2013. Actual sales numbers, however, were comfortably at \$27B (<http://bankinnovation.net/2014/01/paypal-processed-27-billion-in-mobile-payments-in-2013/>) instead. PayPal takes 2.9% of each sale, plus 30 cents. If we estimate the average transaction amount of these sales at a conservative \$25, that means that PayPal made a *surprise extra* \$287M off of those \$7B in mobile sales that outpaced their own projections.

In August 2014, the founder and CEO of Shopify, an e-commerce platform that powers more than \$5B in annual gross merchandise volume, posted this to the company’s blog:

Last week represented the first time in history that more people used mobile phones and tablets to visit online stores than using computers. Looking at data from over 100,000 ecommerce stores that use the Shopify platform, we saw

50.3% of traffic coming from mobile (40.3% from mobile phones, 10% from tablets) and just 49.7% from computers.

TOBI LÜTKE (FOUNDER, CEO, SHOPIFY)

Clearly, web creators can no longer afford to ignore mobile device users, because in doing so they will be ignoring the *majority* of their users.

The Mobile Web Today

In April 2013’s “Communications of the ACM (Association for Computing Machinery),” web technologist Nicholas C. Zakas wrote that mobile phones, at present, were more powerful than the Apollo Guidance Computer used in the 1969 lunar landing of Apollo 11. However, despite their power, the mobile devices of 2013 still suffered from web performance and connection problems similar to those of the early (~1996) stages of the consumer Internet.

Zakas continues, explaining that mobile devices have “high-latency connections, slower CPUs, and less memory” than their desktop counterparts. These devices have slower download request and response times, and need to account for the latency of wireless data transmission.

As a result, we, as developers and designers, must rethink web applications, which were previously geared toward these desktop experiences with wired connections, fast CPUs, and almost endless memory.

Mobile Web Considerations

In terms of building apps for this evolving state of the web, you should keep several considerations in mind. The following sections describe not so much rules or limitations, but rather ergonomic “tent poles” that should guide and support how you craft the user’s web experience.

Small Screen Size

Let's start with the most obvious: Phone screens are considerably smaller than computer monitors. Although it's impossible to pick a *standard* mobile screen size, a good rule of thumb is to start supporting content around 320px (20em at 16px) wide, and grow upward as the content can support it.

Navigation

Coupled with small screen size for any main content is the lack of real estate available for all the navigational tools we take for granted when using big screens.

Compared to a desktop screen, every scrollbar, notification, or menu that remains fixed on the page takes up a much higher percentage of total available space. As such, we must find new navigation patterns to smartly hide content while keeping it discoverable when necessary.

Speed

Mobile devices, when not connected to Wi-Fi, run on far inferior networks than their desktop older siblings. While these devices have improved and will continue to improve, we cannot assume parity between mobile and desktop browsing. As such, large images, heavy JavaScript libraries, or piles of content being forced onto the user will cause painful slowdowns.

People expect their web browsing to be fast. We have gotten used to the speed afforded us on broadband networks. In fact, Nielsen Norman Group research (www.nngroup.com/articles/response-times-3-important-limits/) has shown that:

1.0 second is about the limit for the user's flow of thought to stay uninterrupted, even though the user will notice the delay. Normally, no special feedback is necessary during delays of more than 0.1 but less than 1.0 second, but the user does lose the feeling of operating directly on the data.

Google suggests that people expect websites to load in under two seconds (googlewebmastercentral.blogspot.com/2013/08/making-smartphone-sites-load-fast.html), and that site abandonment rates skyrocket as load times climb

past this number. Speed is actually ranked as the number one reason for abandonment on e-commerce websites.

As an example: I live in Chicago, which is not a tiny city. Twice a day, during my commute, I have no Internet connection while I am underground. Despite this relatively short inconvenience, I regularly notice myself becoming very annoyed that I can't refresh my tweets.

Bandwidth Limitations

When data providers *have* faster networks, it's safe to say that they will charge for them. Access and bandwidth charges (or overages) can be crippling, and, as with speed, every kB you add to your web experience has a tangible cost to the user.

JavaScript and CSS Support

Mobile browsers have gotten much more advanced in the past few years, and a large portion now run a version of Webkit—an open-source browser rendering engine. However, as with *any* web development, testing on a real device is the best way to determine support.

Mobile User Context

Finally, the physical environment of a mobile user is far less standardized than that of a desktop user. In the case of the latter, they are at a desk, or perhaps at a coffee shop, pretending to work on their manuscript.

Conversely, the mobile device user could be anywhere—at that same desk as the desktop user, trying to give directions to a cab driver, checking a map while mountain climbing, or looking up stock quotes while lying in bed.

To wit: When was the last time you went to the bathroom without your phone?

Benefits of the Mobile Web

As you have seen in the preceding sections, there are a number of unique challenges in designing for the web. As more classes of device, more screen sizes, and more

capabilities show up and disappear in devices, it's up to you to decide just how much time and money your team is willing to put into optimization.

Take a breath, though. There is a whole slew of benefits unique to the mobile web. Just as the rise in devices has complicated our lives as designers and developers, it has also unlocked the Internet itself to a massive amount of consumers, and grown the medium into something far greater than the hypertext pages of 1990.

Interaction Support

Let's start with the obvious again: There are many more intuitive, built-in ways to interact with a smartphone than with a computer. Every way that you can touch, pinch, swipe, and tap a screen can be mapped to a different interaction, or behavior.

More important, these devices have been enabling more and more of their hardware to access browsers in recent years, including the camera, the microphone, accelerometers, and notification systems.

Lower Barrier to Entry

Internet-enabled mobile devices are also leaps and bounds cheaper than the total running cost of owning a desktop computer, broadband service, and a place to put said computer with said service.

To that point: In early 2012 I was doing research for a medical software company in rural India. In my two weeks there, I never saw a computer outside of the hotel or one of the hospitals that I visited. However, even in a farming village two hours outside of Chennai, I would have been hard-pressed to find anyone without some level of smartphone, and other members of the trip remarked that 2G service was everywhere.

Mobile User Context Information

Let's now reconsider the mobile user's context. We don't know for certain where they are located. However, that's not relevant to us.

We do know that they are using a device with a smaller screen (where real estate is scarce), somewhere between an edge cellular network and Wi-Fi, potentially dealing with bandwidth premiums. Building their experience, we can start to take all of these limitations as inputs to the user's end goal. We know that they are on a specific website to do a specific task, and we have at our disposal a large variety of interaction and navigational tools with which to help them do that thing as quickly and efficiently as possible.

Who cares if they're in the bathroom?

Building for the Most Users

Mobile device users now represent the majority of *all* users. Those of us who work with data visualization sometimes assume that a large portion of our content can still be tailored to larger, more capable screens, but we do so at our own peril.

The visualizations we create and the web applications in which they live are now consumed on devices from 320px of width to massive 5120px-wide retina screens. Therefore, we have to ensure that our content is consumable on as many devices as possible.

Because it isn't possible to service all of them, it's important to identify the ones that matter most to your own subject. I encourage you to do some tracking of your own website's most frequent user agent and screen resolution ranges. Focus on optimizing for 80% of your user base, or even 50%. It may be smaller than you assume.

Mobile Web Design

For years, web teams have designed and built for the desktop. Mobile, if considered, was an afterthought, or a port of the desktop version. The truth is, this made perfect sense for a long time—users weren't yet there, browsing the web on mobile phones was painful, and carriers controlled access to the web on their devices. What's more,

network speeds were so abysmal at first that it was nearly impossible to load anything more complicated than text.

Over the past few years, however, the web landscape has shifted so dramatically that an awareness of mobile as equal to, or even greater than, desktop means greater opportunities for growth, and a better user experience.

The Way To Build Today

The web has always been a mix of simultaneous (and often competing) priorities: performance, design, marketing, advertising, and SEO. With the mobile web overtaking desktop browsing, a new contender has appeared: device diversity. Table 1-1 displays the fragmentation of mobile device platforms (www.gartner.com/newsroom/id/2335616).

TABLE 1-1

The popularity of mobile device platforms by the end of 2012

Platform	Market Share End of 2012	Shipments in Q3 2012	Installed Base End of 2012
Android	53%	129 million	710 million
iOS	19%	27 million	260 million
Symbian	14%	4 million	190 million
BlackBerry	8%	8 million	104 million
Bada	> 2%	5 million	30 million
Windows Phone	< 2%	4 million	20 million

While there is a clear front-runner here, no single group can be reasonably ignored. Keep in mind, though, that these are global figures. Regional market share can differ immensely. China, for example, is the largest smartphone market in the world, and Android holds a market share of over 90% there.

The pie-in-the sky goal is obvious: How do we create an experience that works for everyone everywhere? The reality: How do we create an experience that works for the most people possible?

The Separate Mobile Web

Another frequently used solution for mobile is to create an entirely new mobile web experience. Instead of scaling down the desktop website, teams create what is known as an m.website.

For some time, this was the standard. It required extra work, but allowed for a uniquely mobile version of a website to become reachable by users. This methodology became so popular, in fact, that the Internet Corporation for Assigned Names and Numbers (ICANN) created the .mobi top-level domain in July of 2005. It was sponsored by some huge names in the web, including Google, Microsoft, and Samsung.

One person rather important to the web expressed the opposite opinion:

It is fundamentally useful to be able to quote the URI for some information and then look up that URI in an entirely different context. For example, I may want to look up a restaurant on my laptop, bookmark it, and then, when I only have my phone, check the bookmark to have a look at the evening menu. Or, my travel agent may send me a pointer to my itinerary for a business trip. I may view the itinerary from my office on a large screen and want to see the map, or I may view it at the airport from my phone when all I want is the gate number.

Dividing the web into information destined for different devices, or different classes of user, or different classes of information, breaks the web in a fundamental way.

I urge ICANN not to create the “.mobi” top-level domain.

TIM BERNERS-LEE

Download the App

Another way to handle the mobile web is to encourage (read: force) an app download when users reach your website on a mobile device.

Apps are a terrific way to utilize all of the rich functionality offered by feature-rich smartphones, and they enable you to leverage all of the hardware available.

However, every mobile operating system requires a separate development team, and 100% overlap in development work for every new feature.

Responsive Web Design

Responsive web design (RWD) is an approach to web creation that aims to optimize the user experience on *any* device without resizing, cropping, panning, or being shunted to an app. A responsive website adapts the layout to the device on which it is viewed using CSS media queries, fluid grids, and flexible multimedia.

Spoiler: The entire next chapter is about this approach.

! EYES CLOSED, EARS PLUGGED

The simplest way to handle any problem is to not handle it. A desktop website with desktop conventions and desktop sizing will show up on mobile.

If you believe that your web application will never (or at least rarely) end up being used on mobile, this is the path you should take.

Also: thank you for accidentally buying this book.

A Taste of Progressive Enhancement

Figure 1-7 shows Tim Berners-Lee's first website once again, but this time on a screen from 2014. As you can see, it looks totally fine—perhaps a bit small, but no content is missing, no styles are broken, and everything is there.

In this case, all we really have is content. CSS didn't exist. JavaScript didn't exist. It's due to this lack of current technology that the website still works on a device so different from what it was designed for. However, we can glean an important lesson from this: The basic need of a user is to read and write to a website. Anything more than this is nice, but not necessary.

This concept is called *progressive enhancement*. It is based on this core idea that the fundamental purpose of a website is to convey its content; everything else builds upon that, enhancing this purpose of meeting a user's "needs."

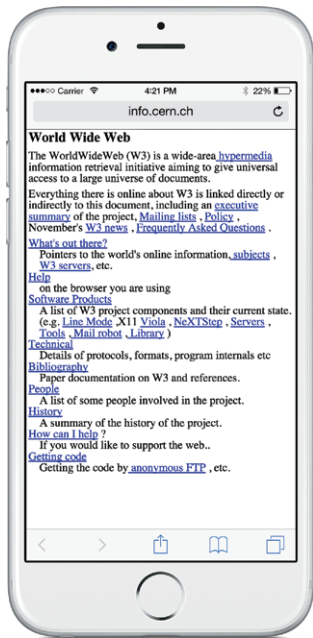


FIGURE 1-7

The first website (1990), displayed on an iPhone (2014)

The absolute minimum experience is always text. There is no technical or design thought given to this level of a website. In addition, HTML is fault-tolerant in various browsers; and as such, the textual content of a website will always be available in this basic form.

The next level of the user experience comes from the page markup itself. HTML semantics provide additional meaning, or meta information, to provide context to the textual content. For example, even though paragraphs and quotes may be displayed in the exact same way, a `<p>` tag represents a paragraph of text to a browser, while a `<quote>` tag represents a quote.

The third level can simply be called design. This is where CSS styling, images, and multimedia come into play. As with HTML, browsers are fault-tolerant with respect to CSS, so they can simply *ignore* rules that are either malformed or that the browser does not support. This is where you can differentiate the style of headings from paragraphs from quotes from links, and so on, thereby turning an *informational* experience into a *beautiful* one.

At the next level we layer on interaction. This is where JavaScript comes into play, and we enhance the aforementioned behavior to clarify, speed up, or delight the end user. Distilled, the principles of progressive enhancement look like this (see Figure 1-8):

- Basic content must be accessible to any browser.
- Semantic markup contains all web content.
- Layout and design are provided by externally linked CSS.
- Behavior enhancement is provided by unobtrusive JavaScript.



FIGURE 1-8

Each layer is built upon the preceding one, with the core idea as the foundation.

Summary

The Internet itself is shockingly old—a military invention from the middle of the last century, but the web that we know today is just pushing 25. That web has come a long way since Tim Berners-Lee’s research in the 1980s and first website in 1990. On the back of this technological shift, entire industries have been born, and with it, the medium of the web has grown tools that let us create complicated, beautiful, and all-inclusive experiences.

All of this history has led to a point at which thousands of different devices can get on the web. For many of these, we can create something unique and beautiful using HTML, CSS, and JavaScript. However, the layers of a web project need to be thought of in exactly that order:

HTML as markup

CSS for design

JavaScript for rich interaction

Interaction is great, but not necessary. Design done right can enhance a point, but is still just fluff. At the core everything still works without any JavaScript, and without any CSS. These are the principles upon which solid mobile web projects are built, and data visualizations as well. For all the exciting toys you have at your disposal now to make rich, beautiful experiences, they are worthless if your users can’t reach the core content you are trying to convey.

02

02

Responsive Web Design Tenets

Before breaking out all of our tools, it's important to set some guiding principles about how to build responsively responsibly. In this chapter, you will examine the foundations of responsive web design, including the guiding principles of your design process—being universal, flexible, economical, and specific. Also covered are seven focal points to build upon as our material becomes more technical.

FYI: This is the content you should casually drop into conversation with someone who can give you a raise.

The Gist

In May of 2010, Ethan Marcotte, a web designer and developer, coined the phrase “Responsive web design.” He did so in the still-accurate and still-important article here: alistapart.com/article/responsive-web-design. Think about that timing for a moment. In the previous chapter, you learned that the iPhone launch drastically shifted the personal computing market in 2007, three years before. And the *mobile tipping point*, that is, the point at which mobile devices began to outsell traditional computers, happened in 2011, one year later.

What device were you using at this point, and how regularly did you access the web on it? Largely, the mobile web was still a second-class citizen at this point, but that was a paradigm that was rapidly shifting. The timing of Marcotte’s article shows that the importance of a multidevice Internet had already begun to take hold, and users were no longer willing to settle for subpar web experiences on their mobile devices.

The concept, or goal, of responsive web design is simple: to design a website in such a way that it provides the optimal experience regardless of a user’s device. A website designed responsively adapts to the device’s viewport so that it appears to be designed for that device. This is achieved with a reliance on proportion-based grid systems, CSS @media queries, and flexible media sizes.

Responsive web design is the practice of building a website that is suitable on every device, and every viewport, no matter the size. It is focused on creating intuitive, natural experiences for everyone so that both mobile and desktop (and any other) users feel like a website was tailored for their device.

There are, in essence, two ways to approach building using responsive web design:

- Build for the limited user.
- Build for every user.

Building for the limited user means focusing mainly on building your core experience and enhancing. Building for every user means leveling the playing field for any class of device, and trying to optimize for as many viewports as possible. These two

mindsets are not at all mutually exclusive. In fact, the first is encapsulated within the second.

That being said, when working on a project, you should consider your audience and your budget. As romantic as it sounds to create a perfect experience for every user, you will find yourself at one of two points:

- Optimizing for just one person.
- Broke.

The important thing is to know the intended users of your product, and optimize, as much as possible, for their use cases.

Building for the Limited User

The first way to approach responsive web design is to build for the more limited user—that is, the user without a large screen, an ergonomic keyboard, strong WiFi, and a desk in a quiet room to contain all these things.

You're now building for people such as those on their morning commute, using a phone that's run out of 4G bandwidth of the month, and whose train is about to enter a tunnel.

While it isn't necessarily accurate that limited users must be on a phone, it's a typical example that serves to illustrate how building for a small device with a potentially poor Internet connection is a good place to start. Think of this method as building *mobile-first*.

Until recently, web designers have approached a project by building out a richly featured desktop site, while leaving the mobile users as a secondary (or skipped) goal. Even as responsive web design has grown in use, often the plan is to begin with a “full” site, and work down by hiding features.

Flip that.

In the previous chapter, you have already seen that the future of the web is mobile, so your design needs to start where your users are. This means beginning with a focus on lightweight, performance-conscious websites built for the mobile user.

Building for Every User

It's important to keep in mind, however, that your users are not *just* on their phones. Once you are accustomed to thinking about your design from the standpoint of someone looking at a small screen in a tunnel, you then have to remember that people will be consuming your content in more locations than desks and tunnels, and on more devices than smartphones. The fact is, we have absolutely no control over where our websites will be consumed.

That's the point of the Internet: It's universal.

This doesn't mean the scattershot throwing of features that work best on a mobile device alongside those that require five screens in a control room at NORAD. What it means is following the same performance-conscious design that you would when building for the mobile user, then taking the time to enhance your website for different classes of device.

Some of the things you may want to consider include the following:

@font-face	CSS Columns	Drag and Drop	Web Sockets
background-size	Generated Content	hashchange Event	Web SQL Database
border-image	CSS Gradients	History Management	Web Workers
border-radius	CSS Reflections	HTML5 Audio	Feature
box-shadow	CSS 2D Transforms	HTML5 Video	Geolocation API
Flexible Box Model	CSS 3D Transforms	IndexedDB	Inline SVG
hsla()	CSS Transitions	Input Attributes	SMIL
Multiple backgrounds	Feature	Input Types	SVG
opacity	applicationCache	localStorage	SVG Clip paths
rgba()	Canvas	Cross-window Messaging	Touch Events
text-shadow	Canvas Text	sessionStorage	WebGL
CSS Animations			

That list contains everything that can be tested for by Modernizr, a tool for feature detection. If it seems daunting, that's because it is. This is typically far too much to consider in any one design (and any one reasonable budget). The important thing to remember is that none of these opportunities for enhancement are contingent upon another; and if you begin in the most accessible state, by building for the limited user, you have already jumped the first hurdle of enabling users to reach your content.

Adaptive vs. Responsive

The term responsive is still in its youth. Adaptive web design (at least as a phrase) can be seen as its older brother. Responsive and adaptive web design are very closely related, and are often used interchangeably.

The design techniques we use are encapsulated within responsive web design, which is a subset of adaptive web design, in order to create designs that dynamically adapt to different viewports and device features. Adaptive design can encompass numerous definitions, and it's important to nail these down so that we can differentiate our more limited definition of *responsive* design.

The Broad Adaptive

“Adaptive Web design,” as I use it, is about creating interfaces that adapt to the user’s capabilities (in terms of both form and function).

AARON GUSTAFSON, ADAPTIVE WEB DESIGN

The term adaptive web design is used in many ways that stretch and broaden its definition. Most accurately, it can be thought of as a big bucket into which we place all of our ways to adapt a website to the user.

Progressive Enhancement

In Chapter one, you were introduced to the concept of progressive enhancement. With its close ties to mobile-first web design, it's easy to see why adaptive web design has been used interchangeably with that term. The basic structure of progressive enhancement is a website with content as its core, with everything else built upon that, progressively rising to meet a user's needs.

At the base level, this is absolutely accurate. You are using a whole slew of client-side and server-side tools to build the best experience across any different environment. This means that interfaces *adapt* to the capabilities of the device and its browser.

Responsive Web Design

Unfortunately, the terms adaptive web design and responsive web design are typically used interchangeably. This is wrong and bad, and anyone I catch doing it will be removed from my Christmas card list.

As Ethan Marcotte defined it in his seminal 2010 article (<http://alistapart.com/article/responsive-web-design>), responsive web design consists of three things: grids, @media queries, and flexible media. That's it.

Creating responsive layouts that work on any device is a crucial part of adaptive design, but it is just one part of a bigger mindset. If adaptive web design is the county fair, responsive web design is the pie eating contest—loud and important, and you'll get your picture with the governor if you do it well enough, but it's not the only part.

RESS

RESS stands for responsive design with server-side components. The bulk of what we're doing responsively will be handled on the client. However, as much as possible, moving solutions to the server can not only make our pages more lightweight, it makes them more automatable. RESS techniques involve offloading some of the responsive work to the backend.

The RESS experience is created through normal responsive strategies, relying on progressive enhancement. The difference is that while under the same URL, the interface is changed in certain ways based on device classes.

At a high level, RESS works by recognizing the device's user agent string (what kind of device it is) and sending that information back to the server when first making contact with the website. The server has a large library of different user agents. Connected to the data of these user agents are lists of properties that are then referenced to determine what type of payload should be given to the client.

We move this logic to the server in order to avoid sending unnecessary (or unused) content to a device. Using a tool like Google's PageSpeed installed on the server, you can set variables from the device's properties, and reuse them to send only the content you need.

As a basic example below, the PHP code pulls in a device library to find the width of a user's device.

```
<?php
include 'DeviceAtlasCloud/Client.php'; // our client library
$results = DeviceAtlasCloudClient::getDeviceData();
                                // fetch properties
$properties = $results['properties']; // store result
$width = (isset($properties['displayWidth'])) ?
$properties['displayWidth'] : ""; // set width
?>
```

Using Google's PageSpeed, you can dynamically replace the image's source with a resized version, as referenced in the code below.

```
"/>
```

You'll notice in Figures 2-1 and 2-2 that the image is exactly the same, but Figure 2-1 is much larger in both actual size and file size. If you were actually trying to show a giant image next to a small one, this would be what you want. However, when working on a responsive design, your media will most often be some fluid size that takes up a percentage of the device's screen. As such, the two images would be *displayed* at the same size while being *downloaded* at the two very different resolutions.



FIGURE 2-1

Without RESS to handle image size, this image is massive.

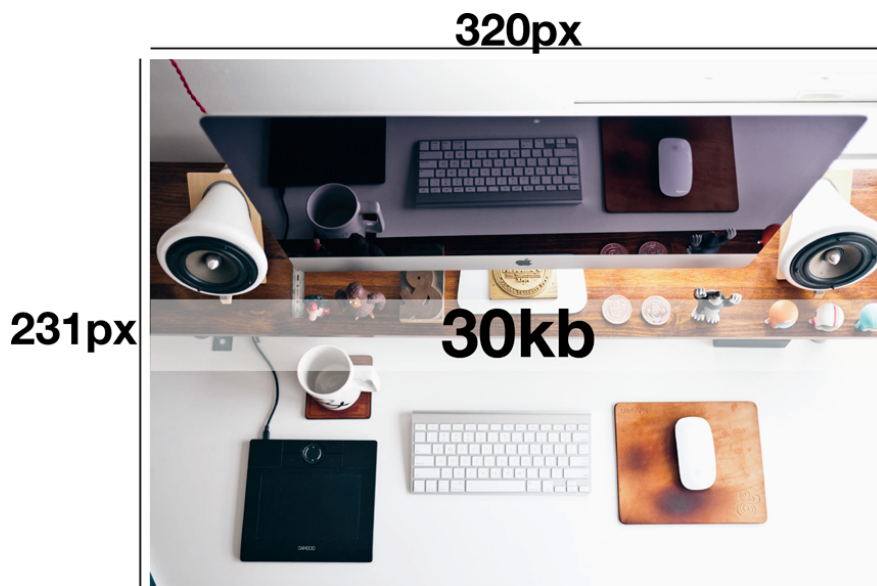


FIGURE 2-2

With RESS, you can cut the image's payload by sending a smaller size from the server.

This same technique can be used to send different CSS, JavaScript, or even direct users to entirely different versions of the page.

RESS is a powerful technique to eliminate bloat, but there is really no reason for it to be confused with adaptive web design. It's simply another tool in your arsenal.

Limiting Your Adaptive Design Definition

Just as “adaptive web design” can be misused in the broadest sense, it can also be misused as very specific concepts. Again, all of these concepts are part of a greater web optimization scheme which we call adaptive web design.

Think of the limited adaptive design definition as a clunky way of trying to optimize for specific devices. Your goal should be to make something work for not just the screens you recognize and regularly use, but for those that you have never seen, and even those that haven't yet been invented. Therefore, I'm presenting the following two ideas: device detection and adaptive layouts, only because they will come up in Chapters 3, 4, and 5. This is not an endorsement of either technique.

Device Detection

Device detection, typically transmitted through the device's user agent string, is exactly what it sounds like: the process of determining an individual type of device and what features it has. While it sounds tempting to start designing by building for an individual device, you then have to build separately for every *other* device as well.

This concept is usefully abstracted with tools like Modernizr, or methods of RESS, by grouping devices into classes that fit into either a yes or a no category for individual features.

Adaptive Layouts

Adaptive layouts are simply responsive layouts without any fluidity. They *adapt* to different viewports with @media queries (or other tactics), but only to defined viewports. There is no leeway or room for growth in between.

For example, let's say you have a website with breakpoints at 480px, 768px, and 960px, the code is below, and the output is shown in Figure 2-3.

```
.content {  
  width: 240px;  
}  
@media screen and (min-width: 480px) {  
  .content {  
    width: 320px;  
  }  
}  
@media screen and (min-width: 768px) {  
  .content {  
    width: 480px;  
  }  
}  
@media screen and (min-width: 960px) {  
  .content {  
    width: 768px;  
  }  
}
```

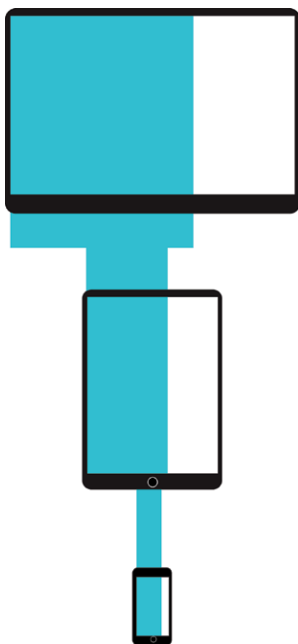


FIGURE 2-3

An adaptive layout at three screen widths between 320px and 1200px

On the other hand, if you were creating a *responsive* layout, the code would look like this, and the output would be Figure 2-4.

```
.content {  
  width: 75%;  
}  
@media screen and (min-width: 480px) {  
  .content {  
    width: 66.7%;  
  }  
}  
@media screen and (min-width: 768px) {  
  .content {  
    width: 62.5%;  
  }  
}  
@media screen and (min-width: 960px) {  
  .content {  
    width: 80%;  
  }  
}
```

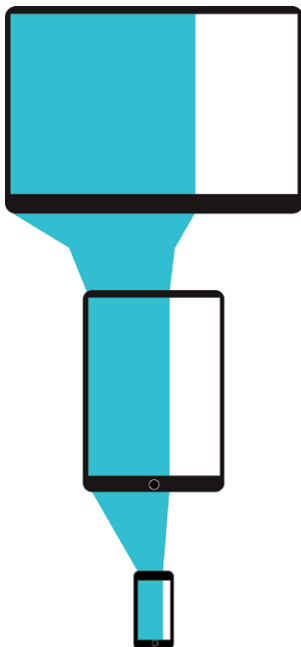


FIGURE 2-4

A fluid responsive layout at three screen widths between 320px and 1200px

The difference is that the adaptive layout is only trying to adapt its fixed-width content at certain specific sizes. The fluid, responsive layout is instead attempting to solve both for those screen sizes that are being called out in the `@media` queries, but also for the spaces in between. If these `@media` queries and breakpoints don't make sense yet, they will by the end of this chapter.

Desktop-First vs. Mobile-First

When you're reacting to different viewports and device features, you still need to start somewhere. Otherwise, you're not responding, you're omniscient. Figure 2-5 shows the conceptual difference in designing desktop-first as opposed to mobile first.

Many of us began our careers in a desktop-only environment. So the natural way to move into responsive design might feel like working desktop-first. To the end user, this may *look* the same, but more often than not, it adds a significant amount of bloat to your pages—especially noticeable to the mobile user.

tl;dr: This is the wrong way.

The code below shows a desktop-first responsive breakpoint.

```
body {  
  background: url("background.png");  
}  
  
/* Media for smaller viewport */  
@media screen and (max-width: 768px) {  
  body {  
    background: #fff;  
  }  
}
```

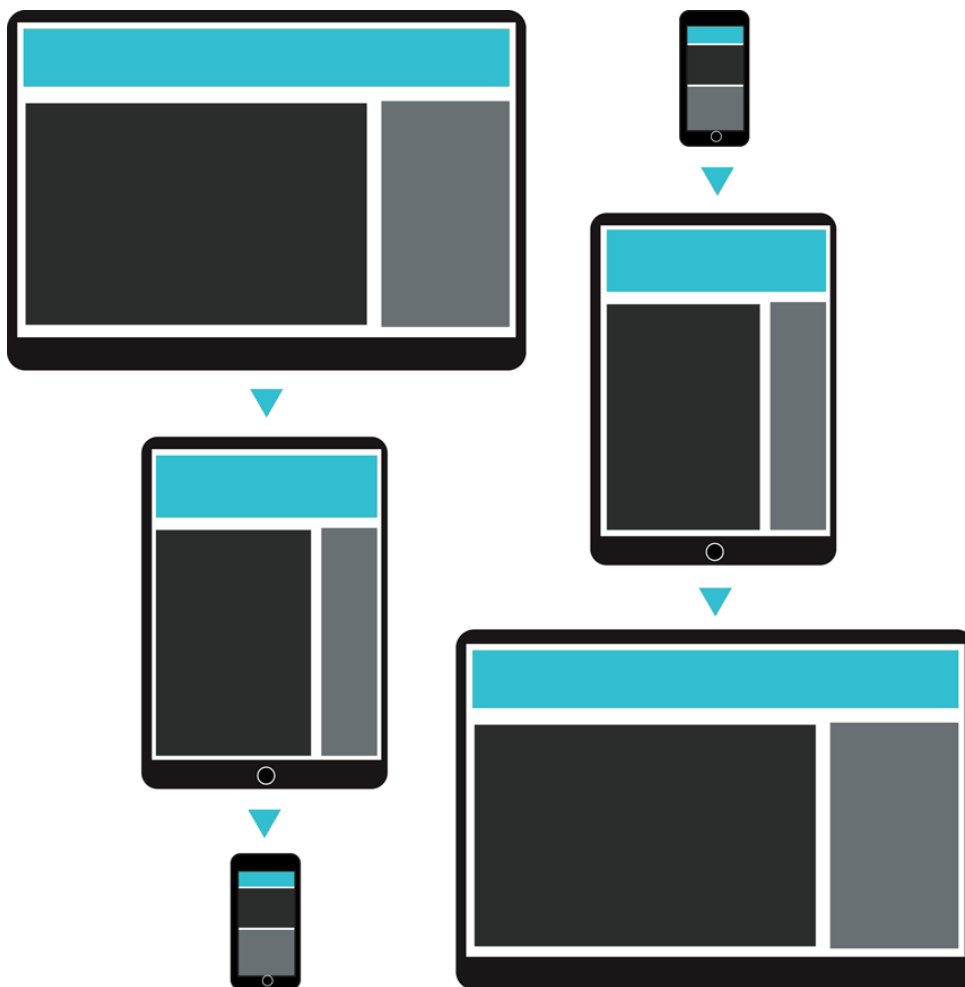


FIGURE 2-5

On the left is the process flow for designing desktop-first. On the right, you end up with a similar product to the end user, but the flow begins mobile-first.

Here, the default style loads an image as a background, which has a high bandwidth cost; but for users on screens smaller than 768px, that image is hidden, overridden with a color. Although the background color might look nicer for mobile users, they have still incurred a totally unnecessary bandwidth cost by downloading the image.

Essentially, what you're doing is building a desktop website, and removing or reflowing content as the viewport shrinks. This goes against the *progressive*

enhancement concept: you are *hiding* things when you don't have functionality, rather than *adding* them when you do.

Mobile-first design, on the other hand, operates on the principle that a user on a mobile device with a smaller viewport and poorer bandwidth should not be required to load the styles and assets for a desktop computer only for them to be immediately hidden or overwritten with mobile styles.

The code below shows a mobile-first responsive breakpoint.

```
body {  
    background: #fff;  
}  
  
/* Media for larger viewport */  
@media screen and (min-width: 768px) {  
    body {  
        background: url("background.png");  
    }  
}
```

In this example, the default case is to set a background color, which is much more lightweight than setting a background image. Only when the user's viewport is 768px or larger do you add the image background. Users on small screens *see* the same experience as our desktop-first example, but get the added performance benefit of not downloading the image.

More important than not causing bloat now, though, is the fact that the desktop share of a user's attention is shrinking. As shown in Chapter 1, the amount of mobile users is quickly overtaking desktop, and building mobile-first means building for the *majority* of Internet users who reach your content from mobile devices.

Four Principles

So how do we guide ourselves through this adaptive world, of which responsive web design is the center tent pole? There is so much more to creating multi-device web experiences than the *tools* of responsive design. To create something that truly

adapts to not only your users' screen, but also to how they naturally *want* to interact with your content, we need a set of principles that we can use to keep ourselves in check, to ensure we're not guilty of the following:

- Doing something just because it's cool.
- Optimizing too heavily for one device.
- Ignoring or alienating an audience group.

The following sections describe the four guiding principles of adaptive web development.

Be Universal

The web, less than a decade ago, was almost universally used on a computer. In addition, it was accessed in a designated space and using a screen that was at least 15".

Now, however, the web is available on a multitude of screens. Therefore, you want to ensure that your site includes as many users as possible.

If people *can* get to your content, you should assume that they *want* to get to it. It's more important to make your website available than it is to make it pixel perfect for a limited number of users.

Content Parity

In an important part of the World Wide Web Consortium's mobile web documentation, the section on "Thematic Consistency of Resource Identified by a URI" officially covers this idea:

This is a realization of the One Web principle, whereby content should be accessible on a range of devices irrespective of differences in presentation capabilities and access mechanism. Web sites may paginate their content in various ways corresponding to differences in device characteristics; therefore the navigation structure of the site, and possibly its technical realization, may vary according to the device class that is being served.

W3C MOBILE WEB BEST PRACTICES 1.0

Which can be shortened to this:

Users want to do everything everywhere. Give them what they're looking for, and don't penalize or pardon them for the way in which they're reaching your content.

Which can then be shortened to this:

There is one web. Don't get in its way.

Forget the Mobile Web

At this point, you can probably guess what all of this means. We've gotten to know each other pretty well at this point, so it's only fair to tell you the entire truth. The key differences between the web and the mobile web are illustrated in Figures 2-6 and 2-7.

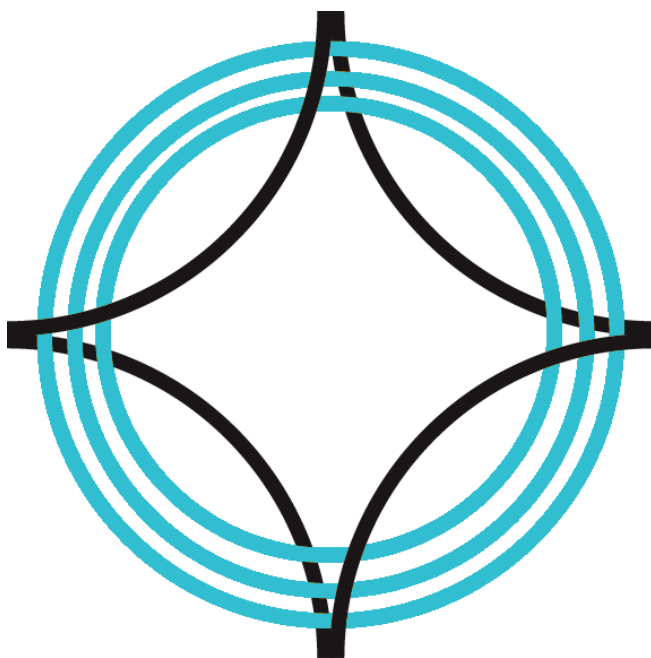


FIGURE 2-6

The web

There is no mobile web.

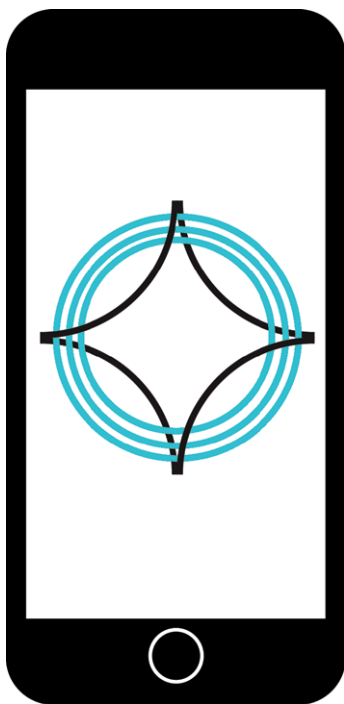


FIGURE 2-7

The mobile web (Spoiler: It's the same.)

It's a myth that mobile users don't want to do everything that desktop users can. Be mindful of this, and don't rely on heavy features that require overly complex technology. Everybody accesses the web in different ways.

Don't lock people out because of their device.

Support and Optimization

Support and optimization were a challenge in web development before mobile browsing came along and complicated things. It's a pain to test across different browsers, and it's a pain to then fix things that are specific to individual browsers. That's why people end up dropping "support" for certain browsers. Support has meant delivering a usable experience, while optimization is spending incremental time to specialize for one platform.

Over the past decade, because a handful of browsers have remained at *similar* levels of usage and popularity in the desktop world, this issue was tackled at an organizational level.

Then mobile browsing showed up, and the amounts of devices (and operating systems and their versions, and browsers and their versions) skyrocketed. When the change was still new, the work required turned into “making a website for iPhone.” These websites also happened to work on Android, because they’re websites, not apps, meaning they work in any browser.

As the web has matured, though, the mobile web experience has continued to become more and more like native app experiences. You have probably come up against this problem before, adding features to your mobile site that make it work, or at least look like it works, like a particularly well-designed iPhone app.

Then you test it on an Android device, and it looks terrible; and then you try to use it on Internet Explorer 8, and your computer catches on fire.

Before web standards existed, sites would often work “best” (read: only) on either Internet Explorer *or* Netscape. Users with the “wrong” browser were asked to switch to the browser for which the site had been built.

This is a perfect analogue to the situation today, except there are far more than two options; and it’s far more expensive to switch between one and the other.

The “Future Friendly Manifesto” sums this issue up perfectly:

Disruption will only accelerate. The quantity and diversity of connected devices—many of which we haven’t imagined yet—will explode, as will the quantity and diversity of the people around the world who use them. Our existing standards, workflows, and infrastructure won’t hold up.

FUTUREFRIEND.LY

That’s not to say that you should kill innovation in your design. Just the opposite: Innovation *breeds* standardization.

Stay *mindful* of different browsers, and let any enhancement you make be exactly that: an enhancement. Build your experience to work, and improve it when you can.

Be Flexible

As you're building for a universal set of devices, you need to begin to let go of some control around exactly what your layout is going to look like. While it is often helpful to build designs that optimize for certain thresholds of screen sizes, you are not going to be able to say that your design will 100% match an Illustrator comp at every pixel size.

That's because you need to stop thinking about pixel sizes. Instead, build with flexible sizes, namely: percentages of the screen real estate. You'll do this by letting the content control the size, shape, and flow of the layout, and simply attaching percentages that work well with that content. In Chapter 3, you'll see just how to move from pixel values to fluid sizes.

From there, you can begin to respond to *changes* in screen size, and optimize where your features and screens demand it. Notice I said features and screens, not devices.

Forget the Device

Just as there is no “mobile web” there is no “mobile-sized web.” We have an ocean of devices to solve for, so let's stop solving for devices.

Organizations and clients, even talented web designers at the leading edge of the field fall into this trap. We create mockups and prototypes and even features that fit into theoretical buckets of mobile, tablet, and desktop, and then congratulate ourselves for solving the Internet.

No, wrong.

Figures 2-8 and 2-9 illustrate the dissonance between how we are wired to think about multidevice web design, and how it exists in reality.

While these broad categories can often be a good *starting point* for your designs, you have to consider the entire spectrum of resolutions that are available now (and will be tomorrow).



FIGURE 2-8

How your brain is trained to think about web design



FIGURE 2-9

Reality

Content in Control

Taking in the massive amount of screens in Figure 2-9, you need to embrace the inherent *fluidity* of the web. You will never know just which screens your content will show on, but you can build for them to work on a whole lot of them. Figure 2-10 shows the first website ever put online.

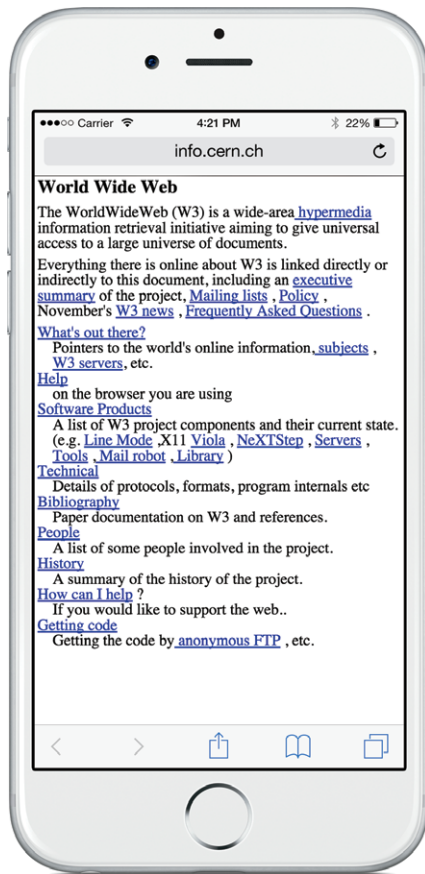


FIGURE 2-10

The birthplace of web design

If you access this site on your phone today, it fits perfectly, and it has zero lines of custom styling. Of course, it also isn't particularly pretty, but it's entirely accessible. You'll want to set breakpoints, change your layout, and use the space you have to make something delightful.

Don't work against the devices your audience might be using. Let your content determine where you set breakpoints. As much as you can, design with *actual content*, and determine through testing what actually works for whatever you're putting online.

Create interfaces that are effective whether the user has 320 pixels or 3200. The goal is to craft web experiences that make it easy for users to interact with the things you're sharing.

Be Economical

Don't bloat your pages. As you'll see once we start exploring our responsive toolset, adaptive web designs can get heavy. This runs exactly counter to the fact that people on mobile devices expect load times that are just as fast as, and often faster than, those available from a desktop.

More complicated still: Your users aren't going to be plugged into an Ethernet cable. As much as we want to see the web as all-permeating, the reality, as recently as 2014, is drastically different just outside of major U.S. cities:

Statistics (from the GSMA) show that only 20% of N. America customers have LTE enabled handsets. Globally, the LTE penetration rate is ~5%. 3G networks are still the norm, not the exception.

BURSTING OUR BUBBLE: MOBILE PERFORMANCE OUTSIDE THE FIRST WORLD

Treat Performance as Design

Our websites are a heavy payload to download. Just as important as appearance is performance.

There are plenty of tools available to find numbers regarding how your site will download, and developer tools to spoof slower networks; but the best way to improve your site-speed is to simply assume it's bad, work on first improving *actual* performance, and then more creatively work in improving *perceived* performance.

Actual Performance

Don't bloat your markup, push images after page load as much as possible, and defer or async your JavaScript.

Look into require.js or another js fetching system, and be sure to minify all your files. Prefetch, preconnect, and prerender your CSS.

Look into basic performance-enhancing techniques such as caching and Gzipping. Consider the speed of your CDN, or, as a first step, use a CDN. Set cache-control and expiry headers in your HTTP.

If some of the preceding terms are unfamiliar at this point, don't worry. We will cover more of these in detail in Chapter 5, but you eager beavers have already put the book down and started searching those terms.

Good, your fingers were sweaty anyway.

Perceived Performance

Just as important as actually having a lightweight, speedy website is having one that *feels* lightweight and speedy. Your website should be like a duck: cool and collected above water, but kicking like mad underwater.

Also, be careful using electronics in the water.

Working in the Browser

Instead of spending hours (days?) on beautiful pixel-perfect comps that will immediately be thrown away once they're actually built, start in the browser. Play around in Creative Suite if you absolutely must, but get to code as quickly as you can.

Not only will you be able to see what works across different screen sizes, but you can also get a feel for the bloat you're potentially introducing with each incremental change.

Be Specific

Finally, remember that you're trying to make something that people want to use. Once you've built something universal and flexible, optimize for the things you know can improve the user's experience. That is to say, once you've spent a lot of time focusing on making something everyone can use, then focus on *who* is going to use it.

You have the opportunity, whether with your own tools, Google Analytics, or another third party, to learn lots of geographical, demographical, and technological information about the people using your website. At the base markup level, you are sending everyone the same content, and they are all hitting the same URI endpoint. But they are doing it in very different ways.

It is up to you to start bucketing your users into those with screen sizes, device features, or even geological regions for which you want to focus additional attention.

The Web Is a Continuum

[This] idea of the Web as a platform, I get why from a marketing perspective, we'd want to use that phrase, because it puts the Web on equal footing with genuine platforms.

I would say Flash is a platform, and native: iOS and Android and these things. They are platforms, in that it's all one bundle. And the Web isn't like that.

What I mean is, if you use the Flash platform, then anyone with the Flash plug-in can get your content. It's on or off. It's one or zero; it's binary. Either they have the platform or they don't. Either they get all your content, or none of your content.

And it's similar with native apps. If you've got the right phone, you can get my app. All of my app. You don't get bits of my app, you get all my app. Or you get none of it because you don't have that particular phone that I'm supporting. And the Web is not like that. The Web is not binary, one or zero, on or off. It's not a platform where you get one hundred per cent or zero per cent. It's this continuum.

JEREMY KEITH, NOVEMBER 2013

The web offers an unprecedented publishing opportunity. Your content can be everywhere; and without over-optimizing for specific platforms, you can optimize for *interaction methods*. That is: Don't work toward something that works on an iPhone, build out the functionality for touch-friendly buttons. Don't force someone to use a logged-in Chrome browser in order to get directions on a map; but recognize if this *is* the case, and layer it on.

Know Your Interface

You can learn vast amounts of information about a machine just by asking. I don't mean "Are you a Nexus 5?" or "Do you have the latest version of Safari?" Rather, think of interfaces in terms of giant Venn diagrams of overlapping *lack of* functionality, and begin to ask devices questions that determine their needs, such as "Are you *not* primarily a mouse-driven device?" or "Are you *unable* to support HTML5 video?"

In the middle of this Venn diagram is your base device, a small brick of a screen that can't do much. Build for this device first, then start to expand into the circles of needed functionality that you have identified, those that will enhance the experience most for your users.

The underlying theme of all these principles? Never assume anything about your users except this: They are trying to do everything everywhere.

Seven Points of Focus

Responsive design, as you're already seeing, involves a large toolset collected to solve the problem of making a design feel *right* on any given screen. To strengthen our footing on this point, we can abstract away the idea of multi-device web design and instead look at the broader history of design for a moment.

Starting with the printed page, designing once meant inches, margins, and layout on paper. (Crazy, right? You're actually looking at that right now.) As screens began to overtake paper as our primary means of media consumption, designers for a long time worked on pixel-perfect mockups that turned into (for the most part) a static website that could be printed onto a page and look about the same as its onscreen counterpart.

However, more devices, and more *sizes* of device, are created by the moment. Layouts can no longer guarantee the size of the paper on which they're printed. Rather

than fighting that reality, though, you can embrace it to develop your understanding of the following focuses: measure, flow, nesting, endpoints, breakpoints, typography, and images.

Measure

The first point of focus, one that gets you immediately 50 steps into a more device-agnostic design, is measure—that is, how you measure the portions of your layout.

In the traditional web design process, you could assume, let's say, 1024px of width. Within this, you could set up a section with 768px width next to one with 256px, and that would be the size of the website. Full stop.

But in our new responsive world, the canvas can be a desktop, a mobile screen, or anything in between. Pixel density can also vary, so you need units that are flexible.

That's where relative units come in handy. The most straightforward one to use is the percentage. Figure 2-11 shows how a layout can be measured first statically (on the left), then flexibly (on the right). At the same 1024px screen, the layouts are identical. The difference is that as the screen size changes, the flexible layout changes with it.

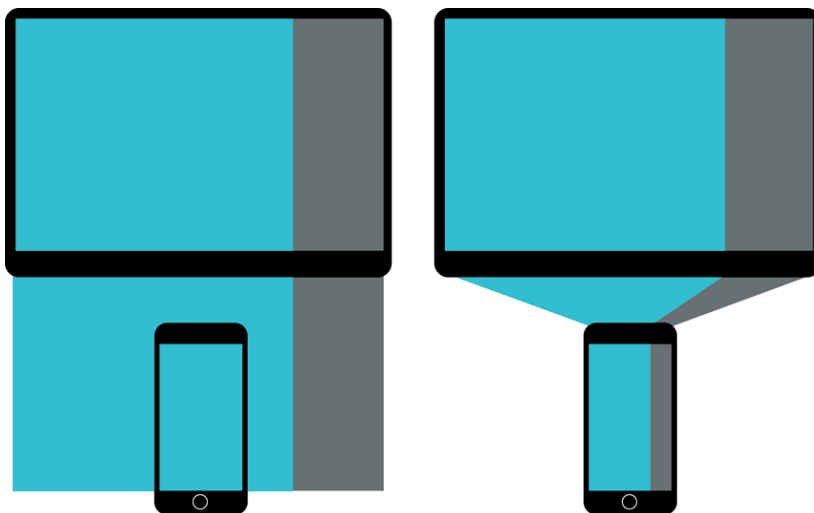


FIGURE 2-11

On the left, a static layout as the screen size changes. On the right, a layout with fluid measure at changing screen sizes

The HTML and static (left) CSS for Figure 2-11 are shown in the following code.

```
<body>
  <section class="section-content">
  </section>
  <aside class="section-sidebar">
  </aside>
</body>

.section-content,
.section-sidebar {
  float: left;
}
.section-content {
  width: 768px;
}
.section-sidebar {
  width: 256px;
}
```

Here, instead, the CSS uses fluid percentages, as in the right side of Figure 2-11.

```
.section-content,
.section-sidebar {
  float: left;
}

.section-content {
  width: 75%;
}
.section-sidebar {
  width: 25%;
}
```

Therefore, making something 50% wide means it will always take half of the screen (or viewport, which is the size of the opened browser window).

Flow

Flow is the way in which content moves within your layout. If you are thinking in a pixel-perfect (and static) world, this may be hard to grasp. Say, for example, you have the layout shown in Figure 2-12, with the markup below.



FIGURE 2-12

A layout built with absolute sizes, on three different sizes of device

```
<body>
  <header class="section-header">
  </header>
  <section class="section-content">
  </section>
  <footer class="footer">
  </footer>
</body>
```

Absolute positioned and sized elements cannot reflow as the browser viewport changes. Following is the CSS.

```
.section-header {
  height: 200px;
  position: absolute;
  top: 0;
  width: 100%;
}
.section-content {
  height: 600px;
  position: absolute;
  top: 200px;
  width: 100%;
}
.footer {
  height: 100px;
```

```
position: absolute;
top: 800px;
width: 100%;
}
```

As shown in Figure 2-13, as screen size becomes smaller, content starts to take up more vertical space and any screen that is smaller than the calculated sizes will overlap and break with the current layout. Instead, you can *remove some rules* in order to let the content flow as needed.



FIGURE 2-13

A layout built with fluid sizing, on three different sizes of device

The code change below actually enables you to remove half of the rules we were applying. In fact, if you removed this code completely, Figure 2-13 would work exactly as expected!

```
.section-header {
  /* height: 200px; */
  position: relative;
  /* top: 0; */
  width: 100%;
}
.section-content {
  /* height: 600px; */
  position: relative;
```

(continues)

(continued)

```
/* top: 200px; */  
width: 100%;  
}  
.footer {  
/* height: 100px; */  
position: relative;  
/* top: 800px; */  
width: 100%;  
}
```

The best part of considering this flow is that it enables you to let your content determine the shape of your layout. Rather than force a box around what your user is trying to access, the information itself drives the website's overall design.

Nesting

The examples presented so far have assumed content that takes up 100% of the user's viewport. Oftentimes, that's not the case. Margins and padding in your designs allow the content to breathe, and make it more parsable for the user. With a flat layout (like the one we're about to construct) this creates a mess of rules that are difficult to control.

The design shown in Figure 2-14 illustrates a layout with the markup below that takes measure and flow into account, giving it some room to breathe by adding margins.

```
<body>  
  <header class="section-header">  
  </header>  
  <section class="section-content">  
  </section>  
  <aside class="section-sidebar">  
  </aside>  
</body>
```

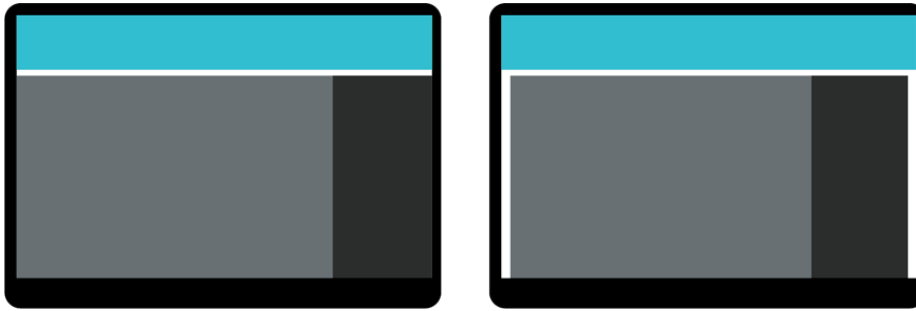



FIGURE 2-14

A layout with no nesting. On the left, there are no margins; on the right, you use CSS to apply margins.

The styling to center the flat layout, and have the appropriate margins in Figure 2-14's right side follows.

```
.section-content,  
.section-sidebar {  
  float: left;  
}  
.section-content {  
  margin-left: 2.5%;  
  margin-top: 16px;  
  width: 75%;  
}  
.section-sidebar {  
  margin-top: 16px;  
  width: 20%;  
}
```

Notice that you are mixing static (pixel) and flexible (percentage) measurements here. This is totally fine, and it is useful for those parts of your layout that you want to keep static, such as the vertical rhythm between sections. That being said, you're introducing a lot of convoluted rules just to get a nicely centered and margined content section.

Figure 2-15 shows the result of markup to nest the main content in a container.

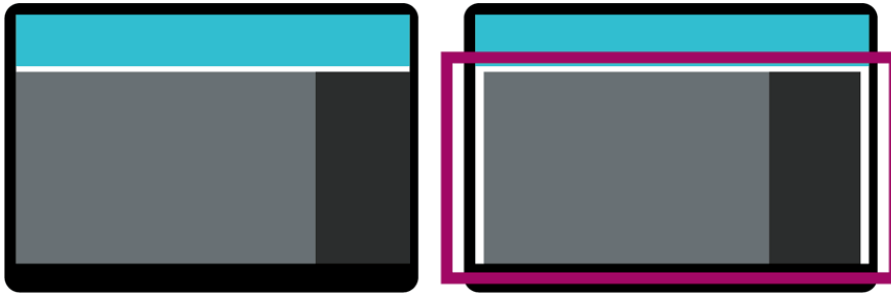


FIGURE 2-15

A layout with nested containers. The HTML and CSS produce the same look. The nested container, although invisible, is highlighted in purple.

The following code shows the new layout for Figure 2-15. Notice the container element wrapping the content and sidebar.

```
<body>
  <header class="section-header">
  </header>
  <div class="container">
    <section class="section-content">
    </section>
    <aside class="section-sidebar">
    </aside>
  </div>
</body>
```

Although the revised styling produces the same layout as the flat version, it is simpler and more maintainable code.

```
.container {
  margin: 16px auto 0;
  width: 95%;
}
.section-content,
.section-sidebar {
  float: left;
}
.section-content {
  width: 75%;
}
.section-sidebar {
  width: 25%;
}
```

As you can see, the effect is exactly the same, while producing much slimmer and more understandable code. Wrapping elements in a container gives you much more straightforward CSS.

Endpoints

Consider the original 1024px static design again. One thing that did solve for nicely was that, although it didn't work on screens smaller than 1024px, it maintained its design on larger screens.

Oftentimes, you want your content to take up the entire width of a screen (especially on smaller viewports), but that same effect usually falls apart on larger screens.

As shown in Figure 2-16, the new markup is nested, flows naturally, and uses flexible measure.

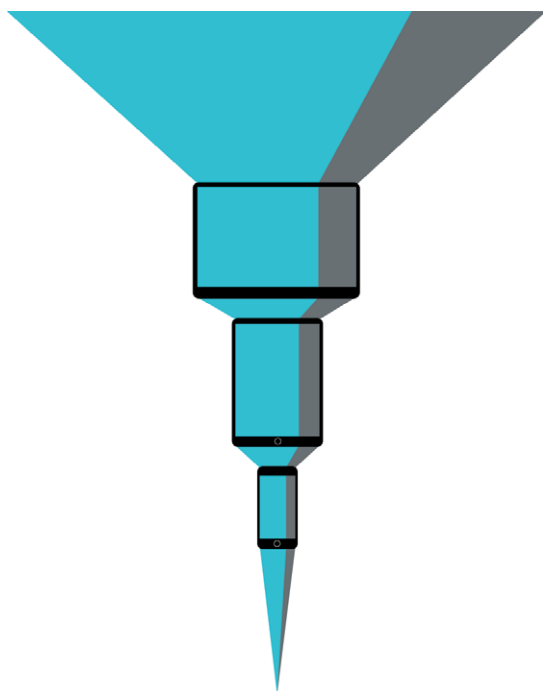


FIGURE 2-16

A nested layout with no endpoints. These sections will scale indefinitely. Notice that the sidebar gets tiny on smaller screens and huge on big screens.

Following is the markup for Figure 2-16:

```
<body>
  <div class="container">
    <section class="section-content">
    </section>
    <aside class="section-sidebar">
    </aside>
  </div>
</body>
```

...and the styling for its layout with no endpoints set.

```
.section-content,
.section-sidebar {
  float: left;
}
.section-content {
  width: 75%;
}
.section-sidebar {
  width: 25%;
}
```

As you can see, on a larger screen, the content just keeps growing. Figure 2-17 shows the result of setting some endpoints in the CSS.

The following CSS shows endpoints applied to our markup, as shown in Figure 2-17.

```
.container {
  margin: 0 auto;
  max-width: 960px;
  min-width: 320px;
}
.section-content,
.section-sidebar {
  float: left;
}
.section-content {
  width: 75%;
}
.section-sidebar {
  width: 25%;
}
```



FIGURE 2-17

Adding a max-width and min-width to the container. This caps the container's width at 960px at the top, and 320px on the bottom, keeping the content and sidebar at a maintainable size.

The max-width rule supersedes the width rule, and the container stops growing once 95% of the screen is 960px. Because the layout is nested, the section-content and section-sidebar stop growing as well.

By adding only one line to the stylesheet, we have essentially capped the width of the container, and now the content simply stays centered once the screen grows beyond ~1010px (960/95%).

Breakpoints

Breakpoints are our first flirtation with one of the major tools you'll be using: the @media query. Breakpoints enable the layout to change at points defined in your CSS. For example, you may want to place all of your content in stacked 100%

width containers on a small screen, but put them in side-by-side containers once a certain size is available.

Figure 2-18 shows the layout we've constructed so far, followed by the markup and CSS.

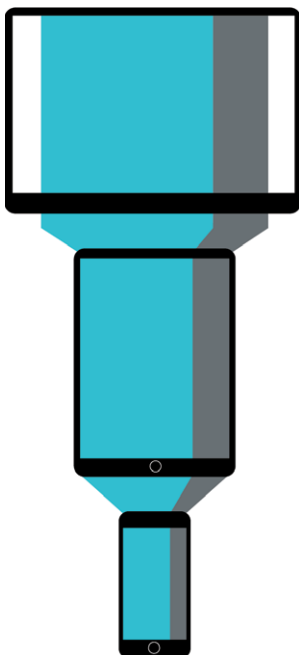


FIGURE 2-18

Our layout without any breakpoints

```
<body>
  <div class="container">
    <section class="section-content">
    </section>
    <aside class="section-sidebar">
    </aside>
  </div>
</body>

.container {
  margin: 0 auto;
  max-width: 960px;
  min-width: 320px;
```

```
}  
.section-content,  
.section-sidebar {  
  float: left;  
}  
.section-content {  
  width: 75%;  
}  
.section-sidebar {  
  width: 25%;  
}
```

On a small screen, the layout isn't stellar right now. That sidebar becomes awfully small.

Now set your *breakpoint* at a min-width of 768px. This means that any rules within your media query will only be applied on a viewport of at least 768px wide, as shown in Figure 2-19.

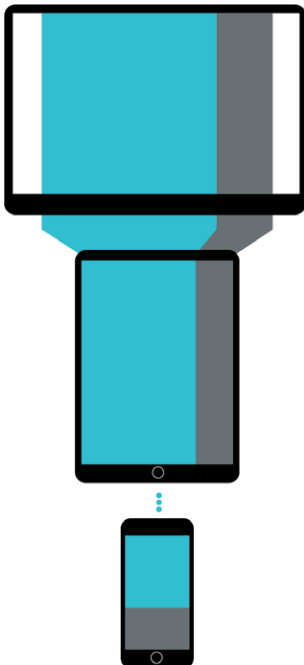


FIGURE 2-19

The layout without a breakpoint at 768px, which is set as the width of your medium-sized device

The code to do just this follows. Notice the `@media` query on line four. All the rules contained within your `@media` query only apply when the viewport is 768px or wider.

```
.container {  
  min-width: 320px;  
}  
@media screen and (min-width: 768px) {  
  .container {  
    margin: 0 auto;  
    max-width: 960px;  
    width: 95%;  
  }  
  .section-content,  
  .section-sidebar {  
    float: left;  
  }  
  .section-content {  
    width: 75%;  
  }  
  .section-sidebar {  
    width: 25%;  
  }  
}
```

Now the layout is a stack of full-width containers that change once the user's screen is wide enough.

Min-Width Versus Max-Width Breakpoints

There are *far* more options than simply min- and max-width when working with breakpoints in CSS; but these are used most frequently simply due to the fact that horizontal real estate is a powerful variable when designing layouts.

- Working with max-width means you're applying more rules as the screen gets *smaller*.
- Working with min-width means you're applying more rules as the screen gets *bigger*.

There is no *right* answer regarding which one you should use. That being said, if you are working mobile-first, it's logical to begin with your base rules at a small screen size, and only start adding rules as more real estate is available.

Put another way: Consider a bigger screen a feature of a device, and progressively enhance your layout as the feature becomes available to you.

Typography

It's easy to forget that the web evolved from print. Long before images and video were so pervasive online, the web was used to share the written word.

There are a slew of books and articles about creating functional and enjoyable web typography—aspects such as typographical measure, line breaks, and font choice constitute a book's worth of content on their own. When we talk about typography as a focus of responsive web design, we are talking about the capability to recalibrate your website's content based upon the user's viewport.

Readability Breakpoints

You're making print content, but you have no idea what size paper will be used, or the DPI of the printer. Let's put some type content into our markup (shown below) for the first time.

```
<body>
  <header class="section-header">
    <h1>My blog post's title</h1>
  </header>
  <div class="section-content">
    <p>Lorem ipsum...</p>
    <p>Lorem ipsum...</p>
    <p>Lorem ipsum...</p>
  </div>
</body>
```

To solve for the virtually endless number of screens on which this content will be viewed, you have two options.

One, you could average out the experience, and create something just okay for every user. The CSS below applies an *average* font size and line height to our layout's typography. Arguably, this is sufficient for all devices but great for none.

```
.section-header h1 {  
  font-size: 48px;  
}  
.section-content {  
  font-size: 14px;  
  line-height: 16px;  
}
```

Two, you can ask your user's device what it can do, and respond appropriately.

Earlier, we used the `@media` query to change the size and layout of our containers, but there's no reason to limit yourself to that. You can just as easily overwrite the rules that apply to your fonts as well. Say instead you use the following CSS. The same `@media` query can be used to specialize the content's font size and line height, not just its container. This enables you to target your content design to a greater level of specificity.

```
.section-header h1 {  
  font-size: 36px;  
}  
.section-content {  
  font-size: 12px;  
  line-height: 16px;  
}  
@media screen and (min-width: 768px) {  
  .section-header h1 {  
    font-size: 72px;  
  }  
  .section-content {  
    font-size: 18px;  
    line-height: 20px;  
  }  
}
```

In this scenario, as a larger screen becomes available, the fonts also become larger to ease readability.

System Fonts and Web Fonts

System fonts, as the name implies, are fonts that are pre-installed on a user's computer. Oftentimes these are referred to as “web-safe” fonts, meaning because they are fonts reasonably installed on every computer, they can be used in web designs with minimal fear that they won't be available to the browser; and because they're already installed, the font loads instantly.

Web fonts are unique fonts available online and served to your website. There are free web font services, such as Google Fonts, that offer a wide array of open-source font options; and fee-based services such as Adobe's Typekit, that offer commercially licensed fonts.

The benefit of these fonts is that you can almost certainly guarantee that the font you have chosen will be the one your user sees, as you are supplying it to their browser, rather than relying on the font already being installed.

The downside is that fonts have weight. With every web font you decide to use, you are adding another file that your user must then download. This can cause a small delay, or flash, of the next available font in your *font stack* before the web font is downloaded.

The font stack is the list of fonts that you have specified in your stylesheet as fallbacks. The CSS specification allows multiple fonts to be listed as fallbacks, as a comma-separated list. It first attempts to load the first font listed, and proceeds down the list until a font is successfully found and used. For example, suppose you have a web font called “MyFont,” which is a fairly standard looking sans-serif for body copy. You would prefer all users to see your content rendered in MyFont, but you can set any number of fallback fonts to render in its place.

In CSS, your font stack looks like the following. This specifies rendering content in MyFont; but if it can't, it will be rendered in Helvetica Neue. If that fails, Helvetica is specified, and so on.

```
.section-content {  
  font-family: "MyFont", "Helvetica Neue", Helvetica, Arial,  
  sans-serif;  
}
```

Web fonts, too, have a clear parallel to progressive enhancement. If your base experience is “sans-serif,” as it is in this case, you can enhance the user’s experience if users have system fonts such as Helvetica or Arial, and then enhance that experience even further if they have the capability and bandwidth to download your web font, MyFont.

As with other features, assume that the ideal font won’t be available, and make sure your content “works” with the default experience. Treat beautiful web fonts as nice-to-have enhancements but don’t, for instance, make a content section’s size contingent upon the letter width of your downloaded font.

Images

Finally, we get to rich media. The conversation about images is not all that different from what we’ve established about layouts. Simply: don’t limit users to static sizes. Your images need to be flexible, such as working in areas of 56% instead of 560px.

The `` tag, as it works today, accepts a height and a width. This is because images were supported on the web before CSS was even invented. You need to forget this fact. From now on, your images are styled by CSS, and you don’t assign them pixel widths.

Rasters and Vectors

You may have noticed two problems with this flexibility:

- Sending a big image to a small screen
- Sending a small image to a big screen

To solve this, you can first look at using a totally different kind of image than what you’re used to.

Rasters are what you likely typically think of when you consider a computer displaying an image. A raster, or bitmap graphic, is a rectangular grid of pixels (or a transparency) stored as an image format such as a .jpg, .png, or .gif. Scaling up a bitmap image causes it to look pixelated or blurry, as you are essentially stretching the matrix to cover more pixels.

Vectors, represented on the web as scalable vector graphics (.svg), instead use geometric primitives (points, lines, curves, shapes, and polygons) to display images. The biggest difference is that these primitives are based on mathematical expressions, and do not map to an individual pixel in a matrix. As such, they can scale indefinitely without looking pixelated or blurry, and (typically) without changing file size.

Consider a simple yellow circle of radius $r=200px$ on a transparent background as an example. To render this as a raster you need a bitmap format that supports transparency (such as .png) the exact size you'll want your image to be (400px x 400px), which has the corresponding matrix of bits drawn to create what looks like a circle at that resolution.

To render it as a vector, you need an indication that what you're drawing is a circle, the radius r , the color, and the location of the circle's center point.

If you then wanted a yellow circle of radius $r=250px$, you could either be satisfied with a blurry image, using the same bitmap as above, or you'd need to create an entirely new raster of 500px x 500px. The vector, however, requires changing only the indicated radius. The image on the left of Figure 2-20 shows a scaled raster image and a vector on the right.

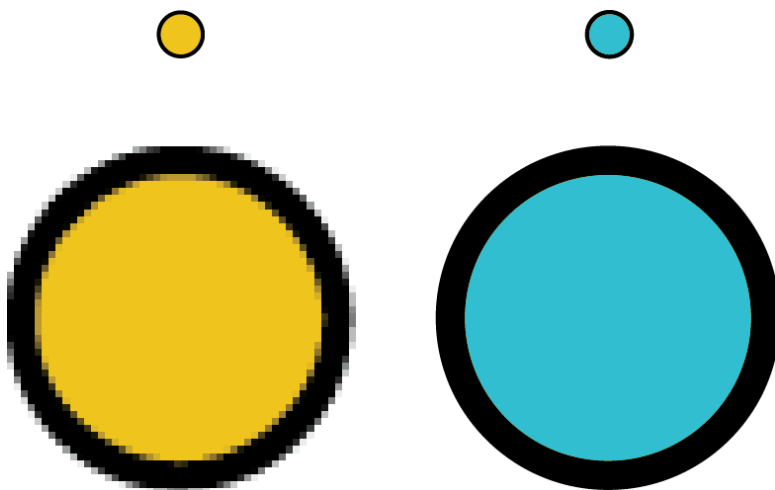


FIGURE 2-20

On the left is an example of raster scaling; on the right, vector scaling. Notice that as large as the vector image gets, no crispness is lost.

Responsive Rasters

It would seem that vectors are the obvious choice for responsive designs, due to their infinite scalability and smaller file size. However, vectors are not suited for all types of images. To actually remain small in terms of payload, you cannot render images at the same level of detail that you can using rasters. The mathematics of the image become too complex and take up more space than the matrix of pixels.

More important, however, not all browsers support .svg images. Blurry images are one thing, but not having an image at all means missing content, and that goes against our stated focus on progressive enhancement.

Instead of relying on vectors to scale for you, you can instead use breakpoints to send different image sizes to different sizes of screen. This works best in your CSS when you are using background-images, as below.

```
#myImg {  
  background-image: url('image1_sm.png');  
}  
@media screen and (min-width: 768px) {  
  #myImg {  
    background-image: url('image1_lg.png');  
  }  
}
```

The downside? At 768px and higher, you are forcing the client to download both images. This solves the *visual* problem, but you take a performance hit. A better solution is to use the RESS (server-side responsive) techniques outlined earlier in this chapter to send images of different sizes to different classes of screen without adding the extra bloat.

The one piece of obvious overhead to this technique is that you must make and store all those different sizes of images. Remember that support and optimization are two different things. At a certain point, you're making bandwidth-heavy, gigantic images for only one user.

Summary

You now have a theoretical understanding of how responsive web design works. In the coming chapters, you will get your hands dirty going through each of the tools that make responsive web design so powerful; but even at this point, you should feel more comfortable understanding what drives an adaptive web mindset, and the reasoning behind it.

Even though we have established that there really is no such thing as a *mobile user*, you can start to build for the *best* experience, which a person on a phone represents nicely.

Be universal.

Be flexible.

Be economical.

Be specific.

The reality in which you're building is one of essentially infinite combinations of user contexts. Screen size is always the most obvious, but you are responsible for making the decision to create something that is both usable in the worst context and delightful in the best. You can embrace this opportunity to develop your understanding of the important areas of focus covered in this chapter: measure, flow, nesting, endpoints, breakpoints, typography, and flexible images.

03

03

Working with a Flexible Grid

The grid is the very base level of responsive design. We tailor our grid to both the viewport's dimensions and the typographic needs of the website in order to create a visual hierarchy, while relying on a mathematical system to keep our design maintainable.

In this chapter, we'll walk through the basics of designing grid systems, exploring a tangible example of creating a fixed-grid system in code, recreating that system with flexible units, and then dipping our toes into creating reusable flexible grids.

The Gist

Grids have informed graphic design in one form or another for a long time, with results ranging from the surprising to the hilarious.

In the mid-twentieth century, graphic designers such as Josef Müller-Brockmann, influenced heavily by modernist ideas put forth in Jas Tschichold's *Die neue Typographie* (Verlag des Bildungsverbandes der Deutschen Buchdrucker, 1928), began to challenge previously held conventions regarding page layout. If you're struggling to envision "previously held conventions regarding page layout," think of a spinning newsreel from a 1930's film, or a busy ad for cigarettes as a cure for polio.

Modernists like Müller-Brockmann, Emil Ruder, and Max Bill began to establish a system for establishing consistency and coherency on a page. In 1961, Müller-Brockmann published *Grid Systems in Graphic Design* (Braun), still considered the seminal book on the topic today (and a darn pretty book). He summarizes the concept nicely:

The grid system is an aid, not a guarantee. It permits a number of possible uses and each designer can look for a solution appropriate to his personal style. But one must learn how to use the grid; it is an art that requires practice.

—JOSEF MÜLLER-BROCKMANN

Oddly enough, the advent of the personal computer, which relieved the need for hand-set typography, spurred a transition away from this style, resulting in experimentation against the grid's rules. However, as the web has become richer, it has become a second home for these typographic and layout tenets.

What Is a Grid System?

Put simply, the grid is a series of vertical and horizontal slices (rows and columns) that serve as a foundation on which a designer organizes modular elements. A grid can be used to group or organize elements within a design, presenting them in a rational and easily comprehended way.

The poster shown in Figure 3-1 embraces the underlying grid system as part of the overall design.

Vermont Symphony Orchestra		
Winter 2007 Season	Aaron Copland The Tender Land January 2007	Eric Satie Gymnopedie 1, 2 February 2007
	01/12/07 Middlebury College Center for the Arts 8:00 pm	02/03/07 Johnson State College Dibden Center for the Arts 8:00 pm
	01/19/07 Johnson State College Dibden Center for the Arts 8:00 pm	02/10/07 Castleton State College Fine Arts Center 8:00 pm
	01/26/07 Lyndon State College Alexander Twilight Theater 8:00 pm	02/17/07 Middlebury College Center for the Arts 8:00 pm

FIGURE 3-1

A poster that includes the grid system as part of its visual design. Visible or not, it would be present.

Why the Grid Is Great

Grids are rad. They enable you to speak a design-development language outside of pixels, determine the balance and positioning of a layout, and provide a foundation for content presentation. In addition, if they are approached in the right way, they can *massively* speed up design and development efforts.

With a well-thought-out grid, the viewer will naturally distinguish between separate modules on a website, and easily identify higher-importance elements simply by whether or not they break from the grid.

However, looking at them through less rose-colored glasses, grid systems serve two distinct purposes in a design: establishing rhythm and defining proportion.

Rhythm

In print and graphic design, and by extension, web design, a grid defines the guidelines by which elements can be positioned in a layout. An effective grid establishes a comfortable meter and rhythm that enables a reader to *access* the content more easily. This rhythm invisibly guides the reader in a way that fosters an understanding of where content can be found within a document.

The grid sets the expectations and rules of the design. Its rhythm is the compass that guides users to where they need to go.

Proportion

The second key role of a grid is to define the proportion of the elements. There is nothing stopping you from randomly slapping unique sizing, margins, and padding to every single element on a page, save for common sense and opposable thumbs.

In print, proportions mainly follow the size of the medium itself—for example, the shape and orientation of a newspaper are reflected in the size of the images and text in its layout.

On the web, this doesn't match as neatly. There is no actual *end* to a web page, vertically or horizontally. Grids still can be anchored back to the viewport, however. Screens are fluid—you have to support as many of them as possible; and by extension, grids can be flexible in order to follow this. You can think of it as making a

layout for any “piece of paper” (i.e., page) the site might need, ever, although that sounds rather daunting.

Grid Systems in Web Design

Built upon the foundations of print, the grid on the web enables designers to set standards for how content is put together on a page. One key difference is that whereas print is obviously limited to a standard page size, grids on the web can be flexible, enabling them to adapt to changing viewports.

Designers like Mark Boulton and Khoi Binh began to explore ways in which this grid system could again bring order to the chaos of layouts.

Grid systems can facilitate creativity by providing a framework and already answer some designers' questions such as “where should the folios go,” “how wide should the measure be?” etc. A well-designed grid system will go some way to answer these questions and more.

MARK BOULTON [www.markboulton.co.uk/journal/
five-simple-steps-to-designing-grid-systems-part-1](http://www.markboulton.co.uk/journal/five-simple-steps-to-designing-grid-systems-part-1)

Flexible Units

Before we get into more complicated implementation details about grids on the web, this section walks through a simple example in order to demonstrate the benefits of working with flexible measurements.

The Meticulous Client

Let's say you're building a simple blog for a client. Let's also say that client is me, and I'm writing a blog about something really interesting: Building Responsive Data Visualization for the Web.

I want a big header on my website’s homepage with a link to that blog post embedded in it. The markup for that, straightforwardly enough, would look like this:

```
<h1>Building Responsive Data Visualization for the Web.  
  <a href="/dataViz " >Read more</a>  
</h1>
```

The result of this markup would be the unstyled content shown in Figure 3-2.

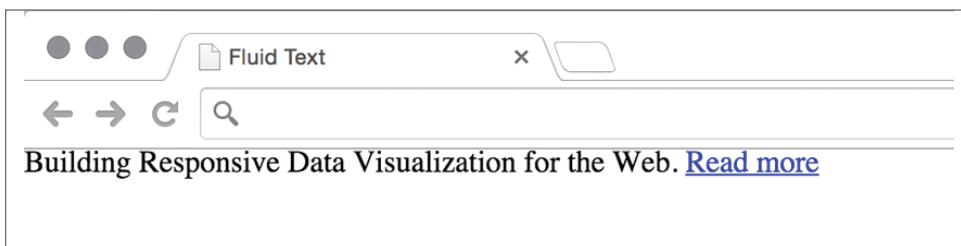


FIGURE 3-2

Unstyled output of the example code

If you’re wondering why the outputted text styles all look the same, it’s because we are using a CSS reset stylesheet. The point of this is reduce browser inconsistencies and give us a uniform starting place upon which to build the styles we actually want without having to assume different defaults per browser.

In this case, we’re using Eric Meyer’s `reset.css` (<http://meyerweb.com/eric/tools/css/reset/>) to default *all* of our styles to a uniform 16px normal weight Times New Roman with no margin or padding.

Technically, this works.

Nonvisual browsers (such as Google) will read the `<h1>` as important content on the page, and the link would take users to `/dataViz` on the same server. If you didn’t care how this looked, you would already have achieved semantic, readable markup. That being said, it’s boring and sad.

I’m a persnickety client, however, and I send the mockup shown in Figure 3-3 to you. Note that it’s a Photoshop document.



FIGURE 3-3

Client's ideal mockup of what the heading and link should look like

When you break out Photoshop and dig through what I actually want these things to look like, you find the following:

- The heading should be 24px bold Helvetica Neue, in an off-black color I defined as #231f20.
- The link, within the heading, should be 13px normal weight italic Baskerville, in a yellow defined as #f1c40f.
- Also, though it's nearly impossible to tell, the background isn't perfectly white, it's #fefefb.

If you don't get these pixel-perfect, you're fired.

Therefore, you start to write the CSS for the `<body>` and `<h1>` tag. It is shown here:

```
body {  
  background: #fefefb;  
}  
h1 {  
  color: #231f20;  
  font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;  
  font-size: 24px;  
  font-weight: bold;  
}
```

The result is shown in Figure 3-4.



FIGURE 3-4

Result after adding styles to the `<body>` and `<h1>`

You're not there yet, in fact we're far from it, but this is a start. The background is slightly less white, and the heading is big. Of course, because you're a good developer, instead of simply setting the font-family to Helvetica Neue, you created a font stack with backups that look similar in case that font isn't available.

The problem should be obvious, though: The link has inherited all the styles of the heading. This makes sense, because the `<a>` tag lives within the `<h1>` tag in the markup. Therefore, you get to work adding styles for the link:



NOTE

For more on CSS font stacks, see the “Typography” section in Chapter 2.

```
h1 a {  
  color: #f1c40f;  
  font-family: Baskerville, "Baskerville Old Face", Garamond, "Times  
New Roman", serif;  
  font-size: 13px;  
  font-style: italic;  
  font-weight: normal;  
}
```

There's nothing too complicated here. You're targeting the `<a>` tag *within* the `<h1>` tag by targeting the CSS rule `h1 a`, then simply following the rules outlined in the design you were given.

After adding that code, the rendered website looks like what is shown in Figure 3-5.

You *could* be done here. The website looks as close to the given design as possible, and there is absolutely nothing wrong with using pixels to define your font sizes. However, in order for this to have any correlation to flexible units, you need to go a step further.



FIGURE 3-5

Finished output in the browser, using fixed-pixel values

Moving to Relative Units

The `em` unit in typography is equal to the inherited point size. On most browsers (and especially after adding `reset.css`), the default `em` size is 16px. That is, 1em = 16px.

However, the `<h1>` is supposed to be 24px, per the comp, and you don't want to be fired just because you're trying to learn about relative units.

If there is only one thing that you get out of this chapter, it should be the following formula:

$$\text{target} \div \text{context} = \text{result}$$

Here's what it means: the *target* font size for the heading is 24px. The *context* is the inherited font size. Because the DOM only includes the `<h1>`, you know that the inherited font size is 16px. Therefore, the following applies:

$$24 \div 16 = 1.5$$

That means 24px divided by 16px is 1.5em. Therefore, you can change the code for the heading to look like the following:

```
h1 {  
  color: #231f20;  
  font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;  
  font-size: 1.5em; /* 24px/16px = 1.5em */  
  font-weight: bold;  
}
```

The output still looks 100% correct, but there's a 13px sitting in the `<h1>` a rule. Try the formula:

$$13 \div 16 = .8125$$

Add that to the code as follows:

```
h1 a {  
  color: #f1c40f;  
  font-family: Baskerville, "Baskerville Old Face", Garamond, "Times  
New Roman", serif;  
  font-size: .8125; /* 13px/16px = .8125em */  
  font-style: italic;  
  font-weight: normal;  
}
```

Figure 3-6 shows the resulting output.



FIGURE 3-6

Output of a first attempt at flexible font-size values. Notice the link size is incorrect.

That's not right. The link is too big, and you're up for a performance review.

If you dig into the pixel value of the link's font-size attribute, you'll see that it's now 19.5px, instead of 13. What happened?

Remember that font size in CSS is inherited from *any* element higher in the DOM tree that happens to change it. The link is a child element of the heading, so it's calculating $.8125 \times 24\text{px}$. What you need to do, in order to get back down to 13px, is change the *context* value from our important equation:

$$13 \div 24 = .541666667$$

That's a gross number. If you're tempted to lop it off at .542ish ems, don't. Browsers love numbers like that—they turn them into nice pixel values of 13. Here's what the CSS for the link now looks like:

```
h1 a {  
  color: #f1c40f;  
  font-family: Baskerville, "Baskerville Old Face", Garamond, "Times  
New Roman", serif;  
  font-size: .541666667; /* 13px/24px = . 541666667em */  
  font-style: italic;  
  font-weight: normal;  
}
```

And your output again looks like the design. Job well done. Even better, the output is now in scalable ems. Why does that matter?

Well, if you're following along with the code, or checking out the demo online, change your `<body>` tag's CSS to the following:

```
body {  
  background: #fefefb;  
  font-size: 20px;  
}
```

You just changed the context for the *entire document*. Rather than leaving the `<h1>` at 24px and the link at 13px, they shifted dynamically to fit their new contexts. Still, though, what's the big deal?

Using a Grid System

You now have an understanding of the history, benefits, and basic building blocks of a grid system. And you've learned the formula to move from fixed to fluid units of measure. You're now going to combine them. Don't worry—this is still going to be baby steps.

You'll first create the measure of your fixed grid, and then move your code from fixed to fluid units. If it isn't yet obvious, the benefit here is that, just as with font

sizes, the fluid units allow your layout to scale to a percentage of available screen space, rather than simply breaking when the screen gets too small, or remaining too small if the screen gets too big.

Let's get into another example.

Another Project

As your client, I liked the work that you did. You've melted my cold heart and I've decided to hire you to redesign my blog; and this time, I don't even have a pixel-perfect design to hold you to. What I do have is a general outline for the content hierarchy, and a weird fascination with this concept of a grid.

Therefore, here's the gist: I need you to make something that has a header section, a content section, and a sidebar section (see Figure 3-7).

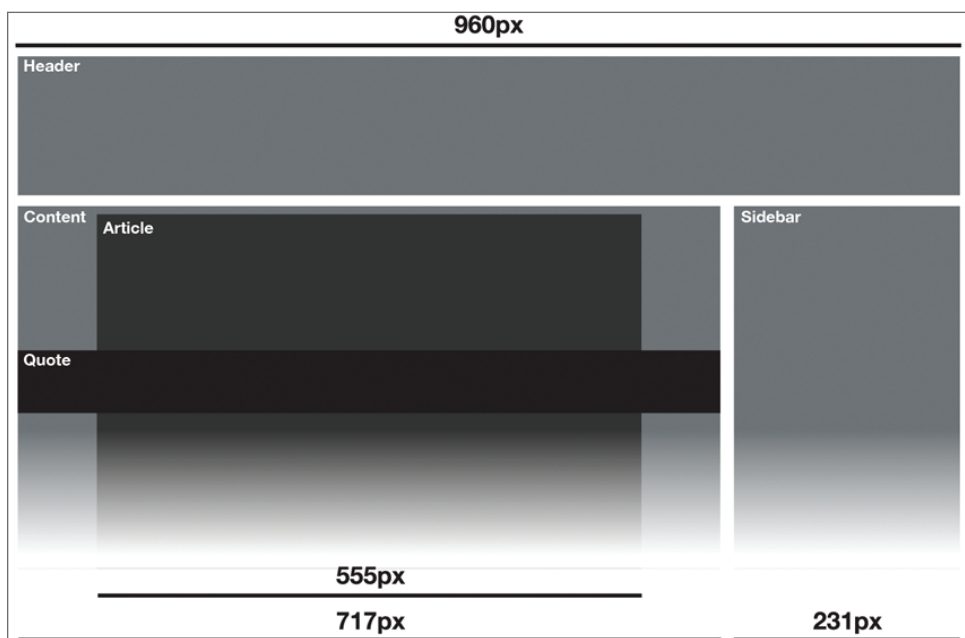


FIGURE 3-7

Layout for the blog project

You're good at this kind of stuff, so you say that, basically, the markup for the page is going to look something like this:

```
<div id="container">
  <h1>...</h1>
  <div id="main" class="content">
    <article class="article-content">
      ...
    <article> <!-- end .article-content -->
  </div> <!-- end .content -->

  <aside id="sidebar" class="sidebar">
    ...
  <aside> <!-- end .sidebar -->
</div> <!-- end #container -->
```

As you look closer at the outline I provided, you realize I was still pretty meticulous, aligning the content to a 960px 12-column grid. You break out the highlighter and come up with the overlay shown in Figure 3-8.

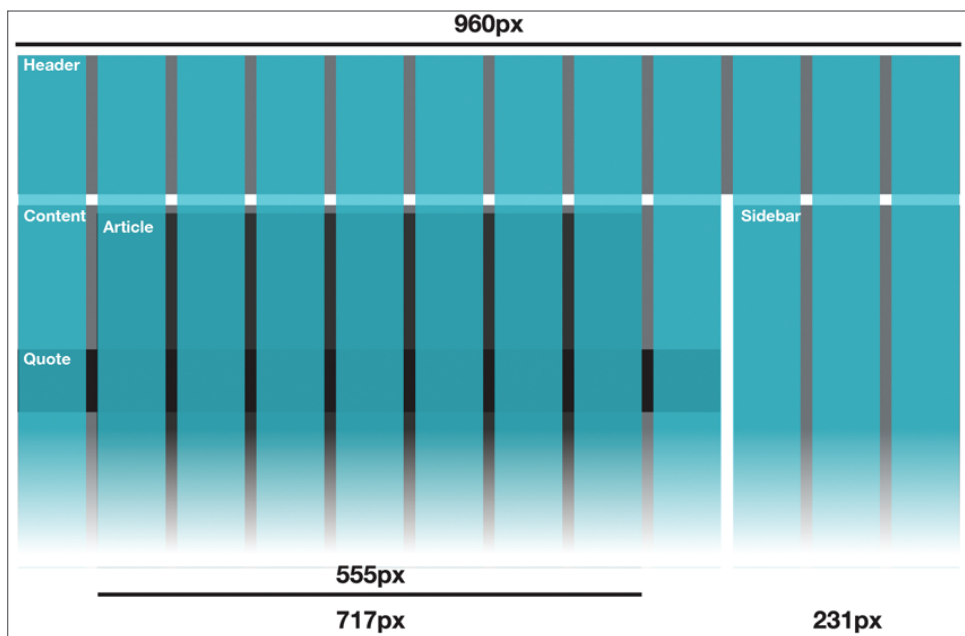


FIGURE 3-8

A 12-column grid overlaid on the blog layout

The grid is 960px total, as marked in the comp; and per the widths of each section, you nail down the unit for each column at 69px, with a 12px gutter in between. The grid (without the content hierarchy) is shown in Figure 3-9.

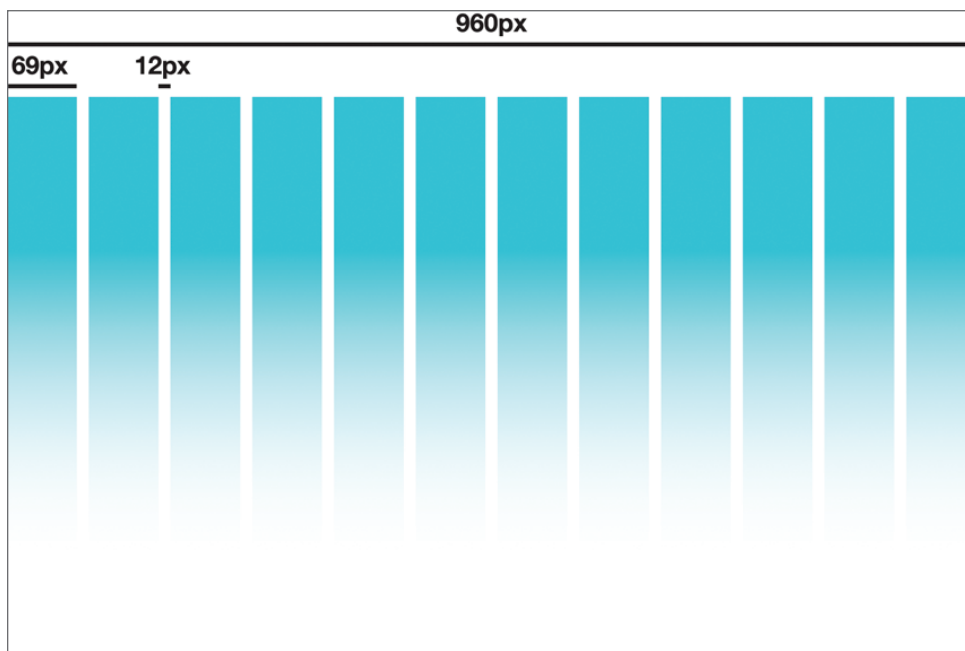


FIGURE 3-9

The 12-column grid, without the layout

From here, you take the reins and start to work with some placeholder content in the browser. I admit that I love the way my current blog looks, but I'm embarrassed that it's not built in a standards-based way, and all the elements are styled with `position: absolute`.

You decide that you're going to make my new blog exactly like the current blog, but with markup that is semantic, and will work outside of the fixed, static screen size, which is shown in Figure 3-10.

In Figure 3-11, you see the same blog with a 12-column grid. As if it was planned as an example in a book, the grid fits nicely over this layout.

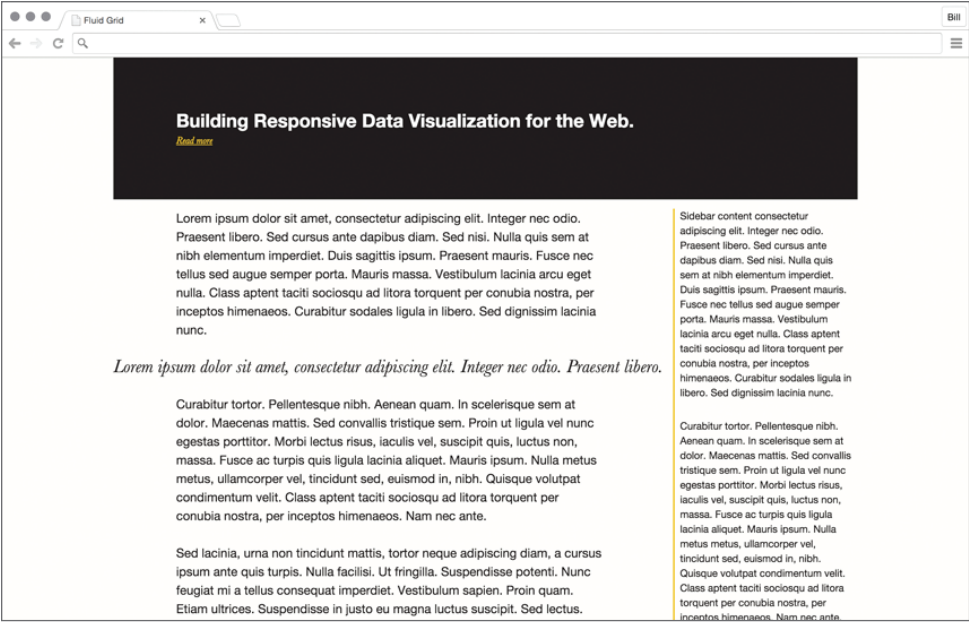


FIGURE 3-10

The current blog design. Who knows what the code looks like?

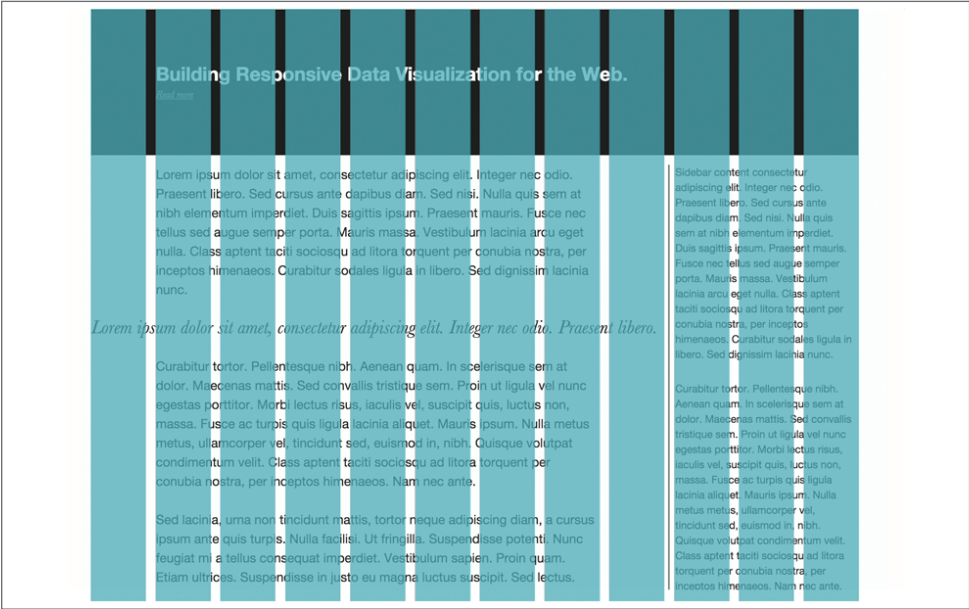


FIGURE 3-11

The current blog design with the same 12-column grid as an overlay

As the content hierarchy is exactly the same as it was in Figure 3-8, you can simply copy over the non-layout styling—that is, the CSS that is simply making things pretty. By this point, the template is starting to take form (see Figure 3-12), and you’re ready to begin chomping away at the layout of the blog.



NOTE

We’re skipping some of the basic CSS rules here. You’ll learn more about a few of them in the next chapter.

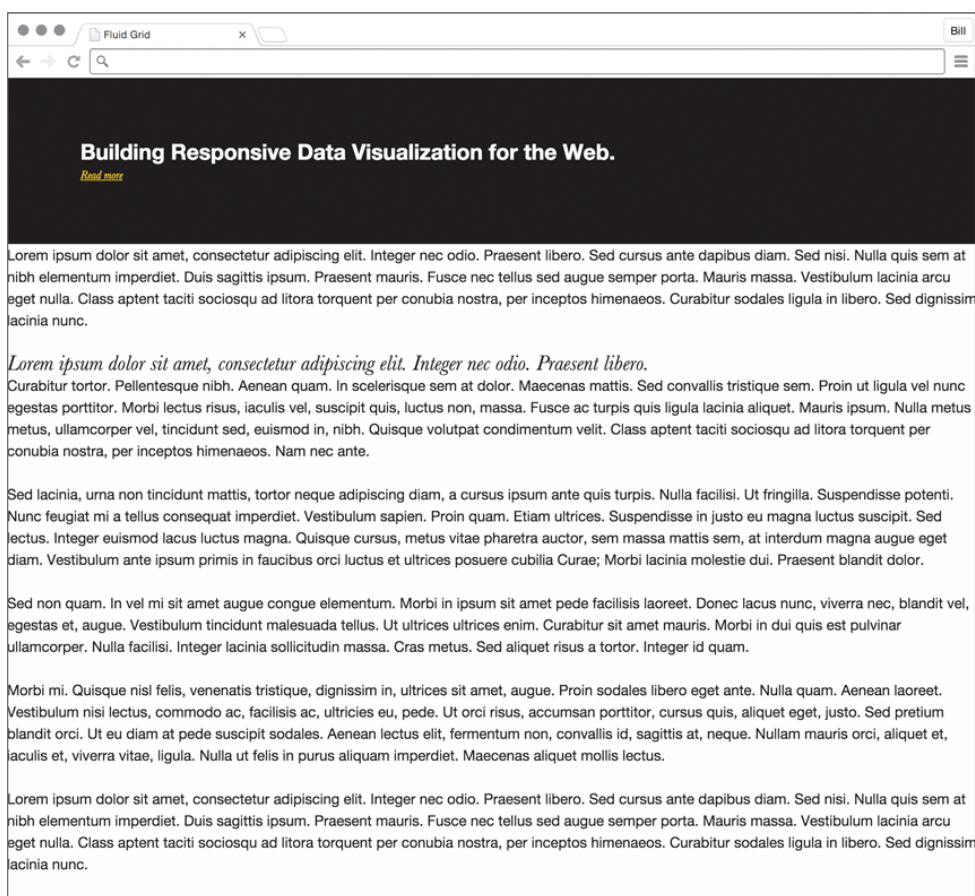


FIGURE 3-12

The blog’s starting point, without any of the actual layout work done

A Fixed Grid

The document I gave you, along with my current blog, had some pretty straightforward section width calculations. We're maxing out the main container's width at 960px for readability's sake. The header spans this full width. The article section takes up the bulk of the main content's width at 717px, with a sidebar on the right for additional content coming in at 231px. There's a 12px gap in the middle of those that adds up to the full 960px (717 + 231 + 12).

I've given this all to you in pixels; and just to placate me, you decide to start by delivering the site back in pixels. Without all of the *actual* design elements, your code looks something like this:

```
#container {
  margin: auto;
  width: 960px;
}
.content,
.sidebar {
  float: left;
}
.content {
  margin: 12px 12px 0 0;
  width: 717px;
}
.article-content {
  margin: auto;
  width: 555px;
}
.sidebar {
  margin: 12px 0 0 -6px;
  padding-left: 6px;
  width: 231px;
}
```

Flex That Grid

You're now once again at a crossroad. The layout matches the comp perfectly, but you don't want to leave your client with an inflexible website. As soon as anyone tries to reach the page on a viewport whose width is smaller than 960px (a great many screens), I'm going to be furious with you.

Therefore, instead of copying the pixel values that I provided in the comp, you decide to move to flexible units. Unlike the typography example, though, you won't be using ems. You'll stick to an even easier to understand measure: percentage.

Building Outside-In

You have to start somewhere, and it might as well be at the biggest element, the `#container`. Previously, it was set at a static 960px. Remember the formula?

$\text{target} \div \text{context} = \text{result}$

But wait, the context in this case is the entire viewport. How can you define a context if that's the exact thing you don't know?

Spoiler: You can't.

What you can do is change what rules you apply to the element. Let's say, for all intents and purposes, you want the `#container` to take up *all* the available screen space, provided it's 960px or smaller. Instead of setting the width to 960px, you recreate that last line of code:

```
#container {  
  margin: auto; /* Same as before */  
  max-width: 960px; /* Max out at 960px */  
  width: 100%; /* Take up all screen space */  
}
```

The Easy Part

Lame. You now have a big element that totally breaks because its children are bigger than it can handle. Moving one layer in, you have the `.content` and `.sidebar` modules to handle. In pixels, they had widths of 717px and 231px, respectively, with 12px between them. With these numbers, you already have your *ideal* context and targets.

For the `.content`, it's $717 \div 960 = .746875$.

For the `.sidebar`, it's $231 \div 960 = .240625$.

Move the decimal places over and you get 74.6875% for the `.content` module, and 24.0625% for the `.sidebar` module. The margin is calculated the same way:

$12 \div 960 = .0125$, so 1.25%

You can see the corresponding pixel and percentage values in Figure 3-13.

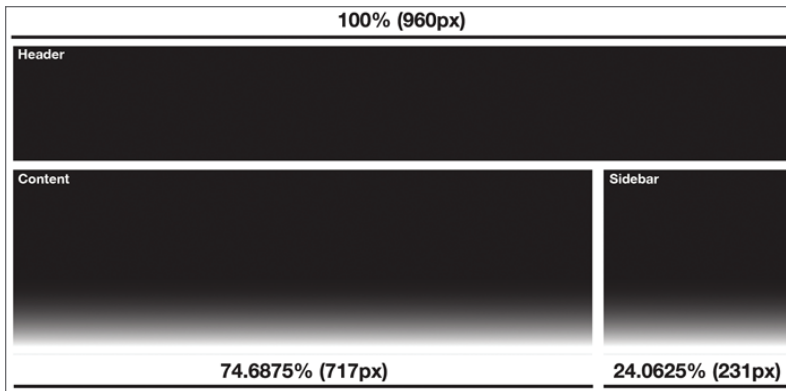


FIGURE 3-13

A little math

Just to confirm, $74.6875 + 24.0625 + 1.25 = 100$. Therefore, your code looks like this:

```
.content {  
  margin: 1.25% 1.25% 0 0;  
  width: 74.6875%; /* 717px / 960px = .746875 */  
}  
.sidebar {  
  margin: 1.25% 0 0 -6px;  
  padding-left: 6px;  
  width: 24.0625%; /* 231px / 960px = .240625 */  
}
```

I still see pixels in there, though. Bummer. The margin is easy enough; you've done that already:

$$6 \div 960 = .00625$$

That works for the padding, too, right?

Don't be silly.

Padding exists within an element, whereas margin exists outside of it, so they must be treated separately. When calculating flexible gutters, the following rules apply:

- **Margin:** The context for this calculation is the width of the containing element.
- **Padding:** The context for this calculation is the width of the element itself.

Therefore, in this case, in order to get an accurate padding-left for the `.sidebar`, the context is actually 231px. That means the flexible padding is:

$$6 \div 231 = .025974026$$

...and the content and sidebar can be written pixel-free as follows:

```
.content {  
  margin: 1.25% 1.25% 0 0;  
  width: 74.6875%; /* 717px / 960px = .746875 */  
}  
.sidebar {  
  margin: 1.25% 0 0 -.625%;  
  padding-left: 2.5974026%; /* 6px / 231px = .025974026 */  
  width: 24.0625%; /* 231px / 960px = .240625 */  
}
```

Changing Context

You're pushing forward to the `.article-content`. Recall from your work with flexible typography that the *context* is determined by the closest DOM parent that defines a rule (just as the link size was determined by the heading size). Therefore, you know that your context isn't really 960px in this case, but actually the 717px of the `.content`. Look at you! You're evolving! Figure 3-14 shows the context calculation.

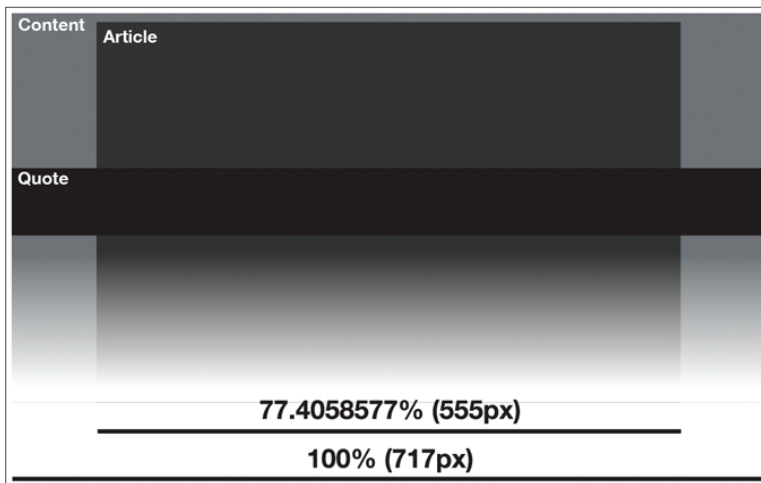


FIGURE 3-14

More math

Only one section left to handle, the `article-content`:

```
.article-content {  
  margin: auto;  
  width: .774058577%; /* 555px / 717px = .774058577 */  
}
```

Negativity

So now we've hit the content section. First, any `<blockquote>`s in the `.article-content` are supposed to take up its full width (see Figure 3-15).

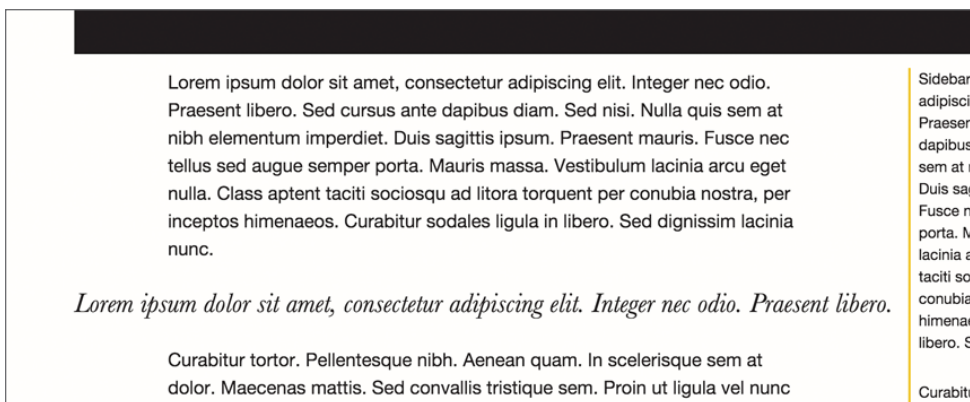


FIGURE 3-15

The quote is supposed to take up the content's full width, but this breaks outside of the article's boundaries.

In the pixel world, that would have looked something like this:

```
.article-content blockquote {  
  margin-left: -81px; /* (717px-555px) / 2 */  
  width: 717px; /* width of .content */  
}
```

But we don't live there anymore. There's nothing unique here. Growing outside of your context works just the same as filling a percentage of it. Your context is the `.article-content`'s width, so 555px. Your target is 717px:

$$717 \div 555 = 1.291891892$$

Therefore, your width is now 129.1891892%.

Remember that margin also relies on the `.article-content`:

$$81 \div 555 = .145945946$$

In code, you end up with this:

```
.article-content blockquote {  
  margin-left: 14.5945946%; /* 81px / 555px = .145945946 */  
  width: 129.1891892%; /* 717px / 555px = 1.291891892 */  
}
```

Revisiting Padding

One last detail.

Your header is 100% width, but the content itself starts with the article content (see Figure 3-16). At this point, you could solve this in your sleep.

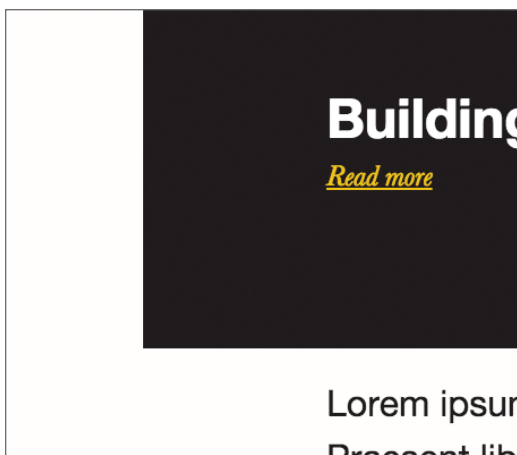


FIGURE 3-16

The header's internal padding keeps it in a column with the blog's main content.

Padding's context is based upon the actual element. In this case, your element is 100% width. At your ideal, that's 960px. In the pixel world, you have 81px to the left of this element, so that's your target:

$$81 \div 960 = .084375$$

```
h1 {  
  padding-left: 8.4375%; /* 81px / 960px = .084375 */  
}
```

That's it! You're done. Code shipped, client happy. It took a whole lot of math, but you've created a 100% flexible grid-based design—one that to someone looking at the old design on the exact same screen would look like nothing had changed.

Branching Out

If you're following along with the code (or have a bit of intuition), you'll notice that this design still totally falls apart on smaller screens. At a certain point, it *doesn't* make sense to have a module on the page only take up approximately 24 percent of the available width. That's around 79 pixels on an iPhone 5. It's an important foundational step, though. You're shifting your mindset to look at the proportions of a website, rather than its exact pixel values.

Creating a Reusable, Flexible Grid (in Five Easy Steps)

Before moving on, I'm going to touch on a key time-saving element of the grid system: reusability. In the exercise you just went through, you had to define the entire grid based on the math corresponding to each element. That's time consuming and ugly and prone to errors.

Although building this grid in code, as opposed to strictly in whatever tool you've used for static mockups, takes a bit of faith between designer and developer (or better yet, designing in the browser itself), it enables rapid movement through the

layout portion of web design work, allowing a greater amount of time and focus to be spent on unique details.

In order to create a reusable grid, you need to define the five elements that make up a grid. Although the elements are presented in a specific order for clarity's sake, no single element can be defined without the other four.

The five elements are:

- Unit
- Gutter
- X-count
- Row
- Column

All five together are visualized in Figure 3-17, and the implementation will become clearer in a moment.

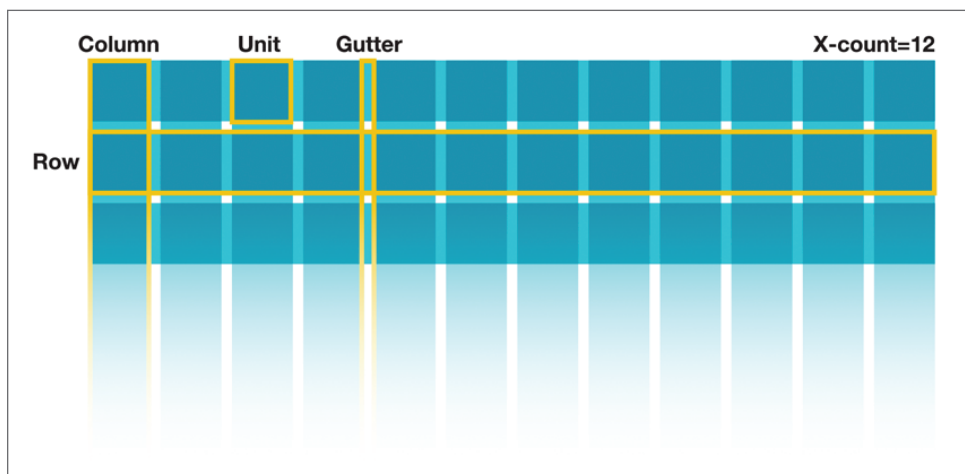


FIGURE 3-17

The five components of a reusable grid

Unit

The unit is the base module for a grid. It's a rectangle that is repeated again and again uniformly to create your grid. Between each unit is a gutter that when combined along the x-axis forms rows, and along the y-axis, columns. In the preceding blog design example, the unit was 69px width by 12px height, although we purposefully did not reference the height, in order to keep the example from getting overcomplicated.

Think of the unit as the `em` of your grid system. Just as (by default) 1 `em` is equal to a rendered value of 16px, the unit is an abstraction of some amount of screen space. It's defined by you to represent one measure of space to be allotted to any given part of a website.

Gutter

The next element of a grid is the gutter. This is the spacing, or breathing room, between each unit. In the blog design exercise, the gutter was set to 12px. Gutters occupy a space between units so that they do not butt up against one another.

In a practical sense, this improves readability and allows for a visual separation of elements on a website. In terms of development, it enables you to define how much “room” you want between things once, then never worry about it again.

X-Count

The X-count is the amount of units (with gutters) you can fit next to one another before moving to a new row. In the blog example, the X-count was 12, as evidenced by the 12 columns.

There are two basic approaches to defining your x-count: content-out and context-in.

Content-Out

If your site contains certain pieces of content that require a defined minimum width, you can determine how many units and gutters you want that minimum

width to take up. By combining unit, gutter, and x-count, you can determine what your grid's full width actually is with the following formula:

$$(\text{Unit}_w) \times \text{x-count} + \text{gutter} \times (\text{x-count}-1) = \text{grid width}$$

Context-In

If you have a specific maximum width and x-count in mind, you can instead derive the unit width by rearranging the formula as follows:

$$(\text{grid width}-\text{gutter} \times (\text{x-count}-1)) \div \text{x-count} = \text{Unit}_w$$

Neither approach is uniformly correct, but by tying your work back to the content defining layout, working content-out makes more sense in our mobile world.

As you can see, all of these elements rely on one another; so at some point you need to make a best guess with at least one in order to define the others.

Columns

The grid's columns are the horizontal regions of multiple units taken together to represent larger parts of the web design. In the blog exercise, the header was a 12-unit column, the content was a nine-unit column, and the sidebar was a three-unit column.

Rows

Hopefully it's obvious, but rows are the vertical regions of units taken together to represent modules in the design. In the blog exercise, the header is a three-unit row, and you did not define the content or the sidebar, as they were longer than the page you were dealing with.

Putting It All Together

You'll begin designing with this grid mindset in the next chapter. Remember how you had to figure out all the flexible element widths for each module in the exercise? Not anymore. Figure 3-18 shows the components of a grid overlayed on your blog project, just as Figure 3-11 showed only columns.

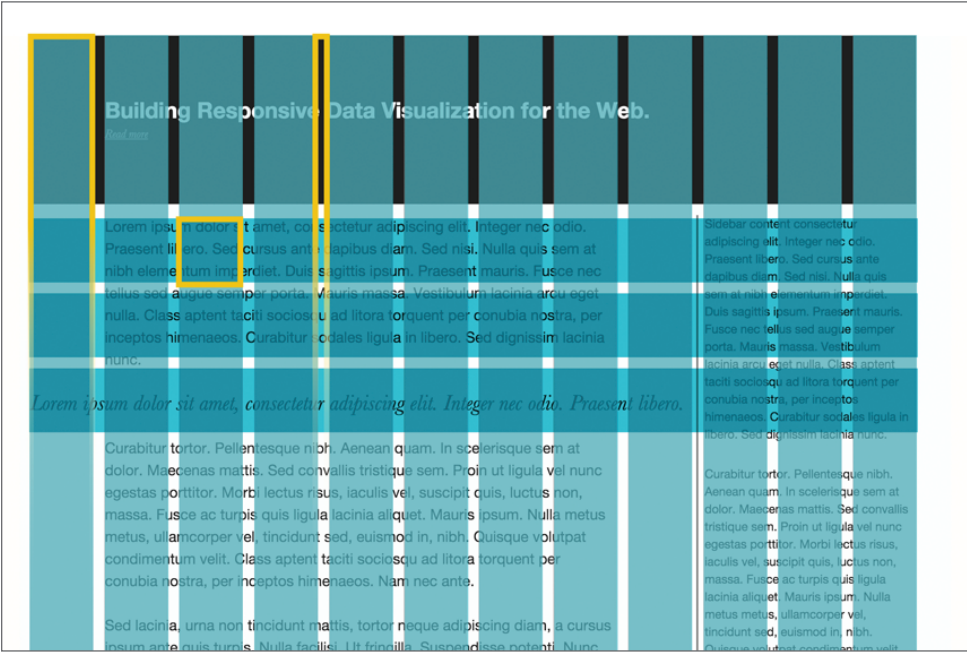


FIGURE 3-18
The five components of a reusable grid overlayed on your previous blog project

Once you separate this layout work from your specific elements, you can apply styles responsively to keep them from breaking at any screen size. More usefully than that, you can turn this grid into a dependency-free CSS module that you can reuse on other projects. This works because none of the styling is tied to *decorative* design, but instead is purely built for layout.

Summary

You’ve now grasped the benefits of a grid system, both in a static layout, and in a flexible one on the web. The components of this grid are: units, gutters, x-count, rows, and columns.

You also now know how to use the formula $target \div context = result$ to move between fixed and fluid units, where *target* is the fixed size of an element, *context* is the fixed size of its container, and *result* is the fluid size that is equal to target in context.

Combining these two, you’re able to create a reusable grid foundation for any responsive web project. What this gives you may not be entirely clear yet, but by the end of the next chapter, you’ll realize that having this standardized base layer for your layout allows you to make changes that *respond* to the user’s screen context much more easily and efficiently.

Try It

Your turn. If you think you’ve mastered the information covered in this chapter, try to make the following slight modifications to what you did.

- **Easy:** Given the following markup, using `ems` *only*, make the `<p>`’s default text 12px, the `<h1>`’s text 24px, and *both* `<a>`’s text 16px:

```
<p>
  <h1>Building Responsive Data Visualization for the Web. <a
href="/dataViz " >Read more</a></h1>
  Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Integer nec odio. Praesent libero. Sed cursus ante dapibus
diam. Sed nisi. Nulla quis sem at nibh elementum imperdiet.
Duis sagittis ipsum. Praesent mauris. Fusce nec tellus sed
augue semper porta. Mauris massa. Vestibulum lacinia arcu
eget nulla. Class aptent taciti sociosqu ad litora torquent
per conubia nostra, per inceptos himenaeos. Curabitur sodales
```

```
ligula in libero. Sed dignissim lacinia nunc.  
  <a href="/somewhereElse">Another link</a>  
</p>
```

- **Intermediate:** Starting where you ended in the blog exercise, change the #container's *ideal* width to 1024px, but keep everything else uniform.
- **Hard:** Using the same markup, change the blog exercise to use a 16-column grid, rather than 12 columns.

04

04

Designing Responsive Layouts with CSS

Building CSS responsively is the linchpin, the idea that shifts our efforts from building flexible websites to building responsive websites. This difference will become clearer as some terms are defined, but think of both fluidity and responsiveness as tools in your adaptive tool belt. Fluidity enables you to grow and shrink your content based upon its container. Responsiveness enables you to ask the container more about its attributes, and change how you handle your content. We're the first apes to touch the monolith.

The Gist

We're officially moving into the *how* of our progressive enhancement tower's second layer: presentation. It astounds me to this day that there was a time when design lived outside of its own bubble, and was a messy, hairy, in-the-way-and-staining-everything part of web markup. You're going to get a brief history lesson into just how design on the web came to be.

From there, you're going to learn the programmatically way in which a browser reads CSS, and how you can query properties of the browser to then begin to target your styling in order to *respond* to different features. At the very end of this chapter, you'll learn a few ways you can cut corners while prototyping, which takes you a few shuffling steps backward to those dark design ages, but first let's establish some context.

A Brief History of CSS

Cascading Style Sheets (CSS) is a stylesheet language. It describes the look and formatting of a document written in a markup language. For the time being, we'll limit the markup we're talking about to HTML, but later on you will dig into using CSS to style scalable vector graphics (SVGs).

CSS is designed to separate the *content* of a document from the document's *presentation*. (Sound familiar?) It is the box in which you contain all of your design; and in the progressive enhancement tower, it sits on top of the content.

This wasn't always the case, though. Looking back at the first website, all that existed was plain text with no decoration, large headings, and underlined links in blue.

As the first decade of the web unfolded, HTML evolved to contain a wider variety of both *semantic* and *stylistic* elements. The issue that arose, however, was that there was absolutely nothing stopping web designers from using one to achieve the goals of the other. That is, although web design has become more robust, it was achieved at the cost of much more complex and inaccessible HTML.

By 1994, the web community had developed a keen interest in improving the presentation capabilities of browsers. At that point, nine different implementations of stylesheet languages had been proposed on the www-style mailing list. Of those nine, two became *actually* influential to what we call CSS today: Cascading HTML Style Sheets (CHSS) and the Stream-based Style Sheet Proposal (SSP).

The CSS Level 1 Recommendation was published by the W3C by the end of 1996. Until this point, development of HTML, CSS, and the DOM had been taken on entirely by the HTML Editorial Review Board (ERB). In early 1997, the ERB was split into three groups tasked with specific projects: the HTML working group, the DOM working group, and the CSS working group.

Not wasting any time, the CSS working group began tackling holes in the CSS Level 1 Recommendation, resulting in the creatively named CSS Level 2 Recommendation in November 1997. This recommendation was published by the W3C on May 12, 1998.

CSS Level 3, which was started in 1998, remains in development to this day. Although browsers regularly implement “proposed” CSS3 features, the spec has not been ratified to the point where it can serve as a definitive W3C recommendation.

Of the more than 50 modules that are part of CSS Level 3, four *have* been adopted as recommendations: color, selectors level 3, namespaces, and media queries.

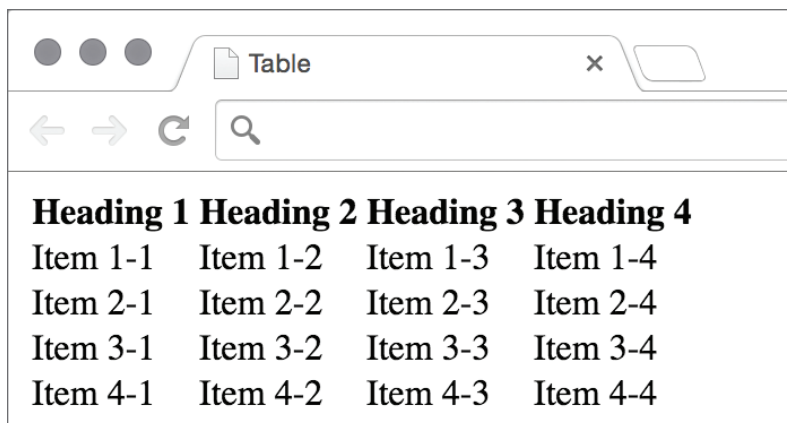
That means the current web design landscape is such that *most* basic rules are implemented uniformly by browsers, while others cannot be counted on.

You’ve Come a Long Way, Baby

While its rocky history might sound like an incredibly depressing way to start a discussion about CSS, don’t worry. I just wanted to get the bad news out of the way first. CSS does provide us, as web designers and developers, with a powerful way to separate content from presentation. What does that mean to an actual human?

You can write markup that is lightweight and semantic, then change the way it looks regardless of how it is marked up. Here’s an example: As the web started to become more of a canvas for design, and not just content, oftentimes web developers would find creative (read: bad) ways to use (read: misuse) the default displays of elements to create (read: shoehorn) a website’s layout. A prominent example of this

is the use of a table. Consider, for example, the default way in which a table is displayed in the browser, as shown in Figure 4-1.



Heading 1	Heading 2	Heading 3	Heading 4
Item 1-1	Item 1-2	Item 1-3	Item 1-4
Item 2-1	Item 2-2	Item 2-3	Item 2-4
Item 3-1	Item 3-2	Item 3-3	Item 3-4
Item 4-1	Item 4-2	Item 4-3	Item 4-4

FIGURE 4-1

Layout of an unstyled table in a browser. Notice that items line up by row and column.

Compared to the HTML elements available in the late 1990s, tables were unique. Other elements were displayed either as a block, taking up 100% of the available width and pushing the next element to a new line, or inline, displayed right next to the next element.

Tables, conversely, included rows and columns that could be used to separate content into modular sections. In addition, tables could be nested within other tables, providing another level of flexibility. Before long, entire websites were built in tables, to get content to display next to, or aligned with, other content.

This behavior, if you can't tell, is absolutely crazy. A table has *important semantic meaning* that completely breaks when it is used for layout purposes. Thinking about a site that aligns content to rows and columns may seem trivial today, but it's thanks to CSS that this is possible.

With properly formed CSS, with efficient *specificity* and *inheritance*, a global stylesheet can be used to easily style elements across an entire website, rather than having to change a property on each instance of the same element. Moreover, an external stylesheet specifies the style once for a range of elements. This is *light years* more performant than repeating style information within HTML.

External stylesheets are also stored in the browser cache, and can be used across multiple pages without being reloaded, reducing bandwidth usage over a network.

How CSS Works

If you're already familiar with how CSS works, good. It is beyond the scope of this book to provide a complete lesson about using CSS in your web development. However, a few basic terms are worth revisiting, as you will build upon them as you move toward adding responsiveness to your web projects.

First and foremost, you should understand the basics of how CSS is written.

Selectors and Rules

Let's say you've written the following HTML for your website; it's a simple `div` with some kind of content inside it, maybe a list of insanely wealthy people from 1909:

```
<div id="titanic">
  ...
</div>
```

You want to style this element with CSS. Because you've given the element an `id` of "titanic," your *selector* will be `#titanic`. The `#` sign represents `id` in CSS.

Within your selector, you can begin to write *rules*, or declarations, of the format:

```
property: value;
```

Here, `property` is some attribute of the element described by the selector, and `value` is what you want that attribute to take on. In this case, you want the `float` attribute of our `div` to be `none`:

```
#titanic {
  float: none;
}
```

In case you missed it, that's the CSS for an incredibly lame joke: The `div` with an `id` of "titanic" has the rule, or declaration, of property "float," which is equal to the value "none." Har har har.

If you want an element to contain multiple rules, separate them with a semicolon and continue to list them, as shown in the following code:

```
<div class="tower-of-pisa"></div>
.tower-of-pisa{
  font-style: italic;
  text-align: right;
}
```

You select the `div` with class of “tower-of-pisa” with the selector `.tower-of-pisa`. The `.` in this case represents class. Then you add rules in the same format. The rules can be listed in any order, but I tend to alphabetize mine for ease of readability and to give the CSS some kind of structure.

Priority

Adding rules to a stylesheet isn’t the only way to create CSS, however; and even within a stylesheet, rules can be written in different ways. Table 4-1 shows the priority structure, from highest to lowest, determining how CSS rules are applied to an element in your DOM.

TABLE 4-1
Priority Levels of CSS Sources

Priority Level	Style Source Type	Style Type Description	Example
0	Importance	Any rule, either in a stylesheet or inline HTML, with the “!important” modifier	h1 { font-size: 24px !important; }
1	Inline	Rules defined in an HTML element’s “style” attribute	<h1 style="color: red;">...</h1>
2	Media type	Rules applied to specific media types as defined in the CSS	@media screen { h1 { color: red; } }

Priority Level	Style Source Type	Style Type Description	Example
3	User defined	A function of the browser, allowing users to define their own CSS overrides	N/A
4	Selector specificity	A specific contextual selector that overwrites the generic definition	<pre>#header h1 { color: blue; }</pre>
5	Rule order	Rules defined last in the CSS are given higher priority than those defined first	<pre>h1 { color: red; } h1 { color: blue; /* This is higher priority */ }</pre>
6	DOM parent inheritance	A property inherited by an element that has been defined on the element's parent	<pre>#heading { font-size: 24px; } h1 { color: red; /* h1 inside #heading will also inherit font- size: 24px; */ }</pre>
7	Generic CSS property	CSS rule or inline style that overwrites a browser default	<pre>h1 { color: red; }</pre>
8	Browser default	The default browser value as determined by W3C initial value specifications	N/A

Specificity

Of course, it would be silly to assume that your DOM will only ever be simply one `<div>` within the `<body>` tag. Elements have parents and children, and your DOM can get very complex as you build your pages. CSS has rules for dealing with the specificity of tags.

Consider the following code:

```
<div class="foo">
  <a href="#">A link</a>
  <a href="#" class="bar">A link with class bar</a>
  <a href="#" class="bar" style="color: red;">A link with class bar
and inline style</a>
  <h1>This is an h1. <a href="#">Link in the h1</a></h1>
  <h1>This is an h1. <a class="bar" href="#">Link in the h1 with class
bar</a></h1>
</div>
```

Notice the following elements in the preceding example:

- **Line two:** A link within the `div` with class `foo`
- **Line three:** A link with class `bar` within the `div` with class `foo`
- **Line four:** Identical to line three, but with inline CSS
- **Line five:** A link within an `h1` within the `div` with class `foo`
- **Line six:** A link with class `bar` within an `h1` within the `div` with class `foo`

Without any external CSS, the only element that would receive any unique styling would be line four, because you are applying a style inline.

The following applies these CSS styles to the HTML above:

```
.foo a {
  color: purple;
}
.foo .bar {
  color: green;
}
.foo h1 a {
  color: blue;
}
```

The output is shown in Figure 4-2.

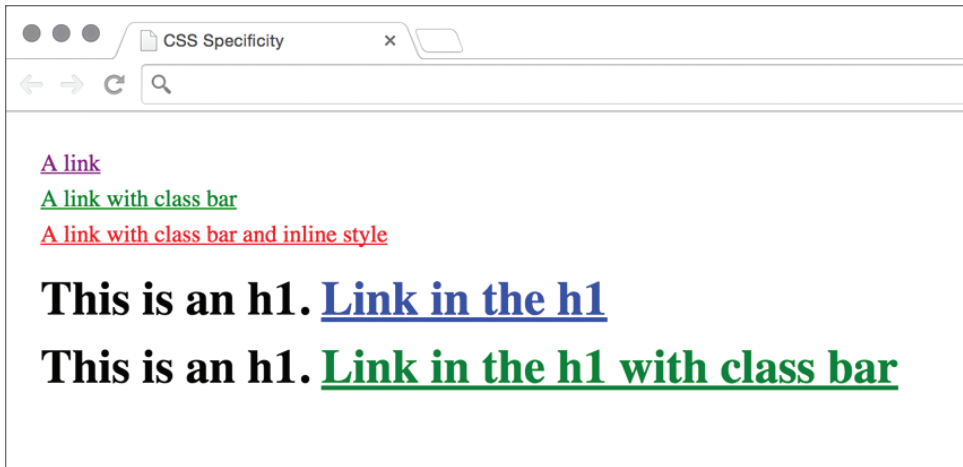


FIGURE 4-2

Output of the styled list of links. Priority and specificity are in effect here.

You see that the classless `a` tag is purple, and the `a` tag on line five is blue. The inline-styled link is still red, and the link outside of the `h1` with class `bar` is green. Were you expecting the link on line six to be blue as well? Although it would make sense that the `.foo h1 a` *seems* more specific than the `.foo .bar`, classes are treated with a higher level of specificity than element tags.

There are plenty of other rules like this, but this is enough of a primer to move forward.

Inheritance

You saw inheritance in effect in the last chapter. Remember the initial flexible typography exercise?

```
<h1>Building Responsive Data Visualization for the Web.  
  <a href="/dataViz ">Read more</a>  
</h1>
```

Given the preceding markup, your output looked like what is shown in Figure 4-3.



FIGURE 4-3

Unstyled markup of the preceding code

However, after you added the following first set of rules, it changed to what is shown in Figure 4-4:

```
h1 {  
  color: #231f20;  
  font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;  
  font-size: 1.5em; /* 24px/16px = 1.5em */  
  font-weight: bold;  
}
```

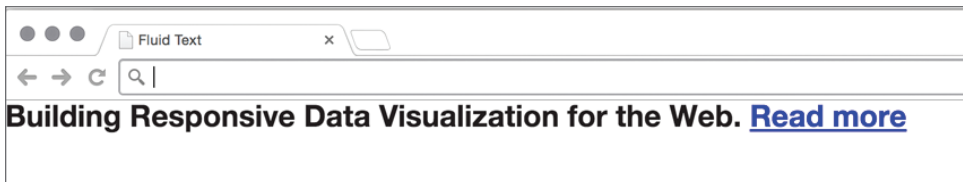


FIGURE 4-4

The same markup, after the preceding styles are applied. Note that the link inherits some, but not all, styles from the heading.

Not only did the font attributes of the `h1` change, but so did the `a` within it. The `a` tag *inherited* the rules applied to the `h1`. However, it did not inherit the color change. That's because color is specifically called out to be different for `a` tags by browsers, so a rule must be specifically crafted for `a` to change that attribute.

Dipping Your Toe In

Now that you have the basics down, you'll look at a more challenging example that combines a few of the concepts. Suppose you have the following markup—a

container with a heading, a link, and a paragraph of text, something like this (Figure 4-5 shows what it looks like in the browser):

```
<div id="container">
  <h1>Responsive Data Visualization. <a href="#">Read More</a></h1>
  <p>...<p>
</div>
```

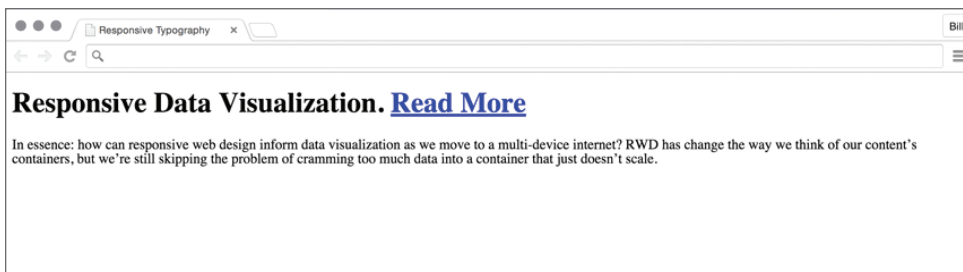


FIGURE 4-5

Unstyled output of the preceding code

Nothing unexpected so far. Moving to larger screens, the mockup (PSD) for what the blog should look like on screens near 1200px wide is shown in Figure 4-6.

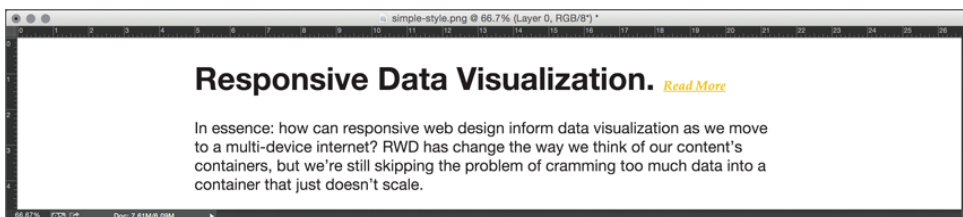


FIGURE 4-6

Designer's mockup of what your blog should look like on a large screen

You want to add a CSS reset stylesheet (like meyerweb.com/eric/tools/css/reset/) to get rid of the default styling for elements, and then start writing your CSS. Opening the design spec, you see that the container is 1200px, so you can immediately

add what you learned about grid flexibility from the last chapter, specifying that the container have a width of 100% with a `max-width` of 1200px:

```
#container {  
  margin: auto;  
  max-width: 1200px;  
  width: 100%;  
}
```

For the rest of the content, first write it in a wasteful way. The `<h1>`, `<a>`, and `<p>` tags all have font sizes and layout elements defined in the PSD that result in the following code and the output is shown in Figure 4-7.

```
h1 {  
  color: #231f20;  
  font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;  
  font-size: 64px;  
}  
h1 a {  
  color: #f1c40f;  
  font-family: "Book Antiqua", Palatino, "Palatino Linotype", "Palatino  
LT STD", Georgia, serif;  
  font-size: 26px;  
  font-style: italic;  
}  
p {  
  color: #231f20;  
  font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;  
  font-size: 32px  
  margin-top: 16px;  
  padding-bottom: 32px;  
}
```



FIGURE 4-7

Your code's output, which now matches the design. Hooray!

You've matched the PSD, so job well done. However, it's not flexible yet. Time to clean up your code. You know that you can determine the flexible sizes with the following formula:

$$\text{Target} \div \text{context} = \text{result}$$

Your base context is 16px for the document, so you can proceed from there.

You also know that you will *inherit* rules from the parent elements, so your font size and color can be added to the container `div`, and removed from the `<h1>` and `<p>` tags' rule declarations:

```
#container {
  color: #231f20;
  font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
  margin: auto;
  max-width: 1200px;
  width: 100%;
}
h1 {
  font-size: 4em; /* 64px/16px = 4em */
}
h1 a {
  color: #f1c40f;
  font-family: "Book Antiqua", Palatino, "Palatino Linotype", "Palatino
LT STD", Georgia, serif;
  font-size: .40625em; /* 26px/64px = .40625 */
  font-style: italic;
}
p {
  font-size: 2em; /* 32px/16px = 2em */
  margin-top: .5em; /* 16px/32px = .5em */
  padding-bottom: 1em; /* 32px/32px = 1em */
}
```

Still not bad, but who says that the base size *has* to be 16px? That's just the browser default, and per Table 4-1, browser default is at the absolute bottom of the priority list.

Looking back at the font sizes, they're mysteriously all nicely divisible by 16, almost as if they were set to show off exactly this point. The document's base size *should* mean something to your content; and in this case, the default content really is the

body copy within your post, which you want to be 32px. Therefore, set your container's font-size to 2 em.

This way, the new context for your page is actually a base of 32 px. Check out the following code:

```
#container {
  color: #231f20;
  font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
  font-size: 2em;
  margin: auto;
  max-width: 1200px;
  width: 100%;
}
h1 {
  font-size: 2em; /* 64px/32px = 2em */
}
h1 a {
  color: #f1c40f;
  font-family: "Book Antiqua", Palatino, "Palatino Linotype", "Palatino
LT STD", Georgia, serif;
  font-size: .40625em; /* 26px/64px = .40625 */
  font-style: italic;
}
p {
  /* No font size rule needed */
  margin-top: .5em; /* 16px/32px = .5em */
  padding-bottom: 1em; /* 32px/32px = 1em */
}
```

It might not seem terribly useful when you're looking at the code. It's still the same number of lines, they're just shifted around a bit; but let's change the viewport through which you're looking at the output. Figures 4-8 and 4-9 show the same page on a tablet-size screen and phone-size screen, respectively. Notice anything wrong?

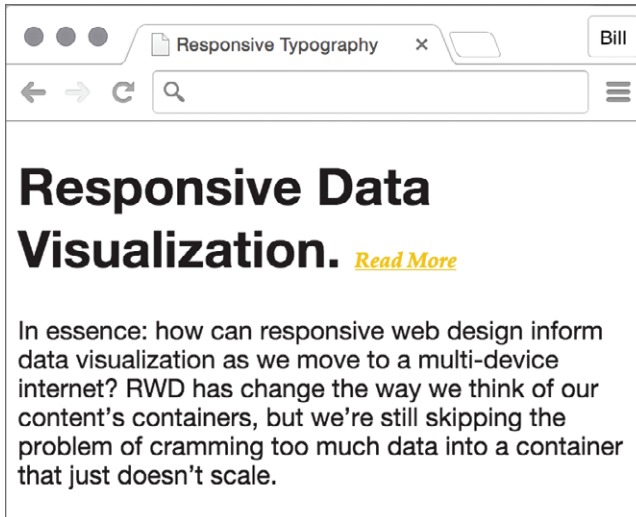


FIGURE 4-8

Your page on a tablet-size screen



FIGURE 4-9

Your page on a mobile-size screen

The visual hierarchy is an absolute mess. The big bold header breaks onto different lines and flows off the page, and the body copy is cut into such short lines that it looks more like a poem than a blog post.

However, you've already written your CSS flexibly, so all the styles rely on their context, right? Therefore, let's change something:

```
#container {  
  font-size: 1.5em;  
}
```

This changes your highest parent's font-size from 2em to 1.5 em, meaning $1.5 \times (16\text{px}) = 24\text{px}$. Figure 4-10 shows the result on the tablet-size screen.

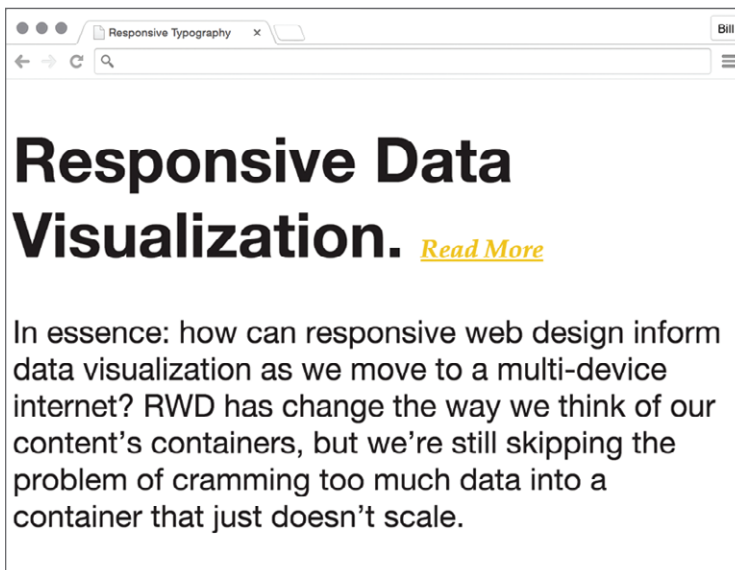


FIGURE 4-10

Your page on a tablet-size screen after changing the base font size to 1.5 em

Now take it another step down for the mobile screen. Instead of 2 or 1.5 em, set your base size to 1 em. Therefore, $1 \times (16\text{px}) = 16\text{px}$. (Extra points if you are thinking “Why not just get rid of the rule entirely?” I’m just trying to prove a point!) Figure 4-11 shows the page on a tablet-sized device after making the change in the code below.

```
#container {  
  font-size: 1em; /* 1em means this rule could be completely removed */  
}
```

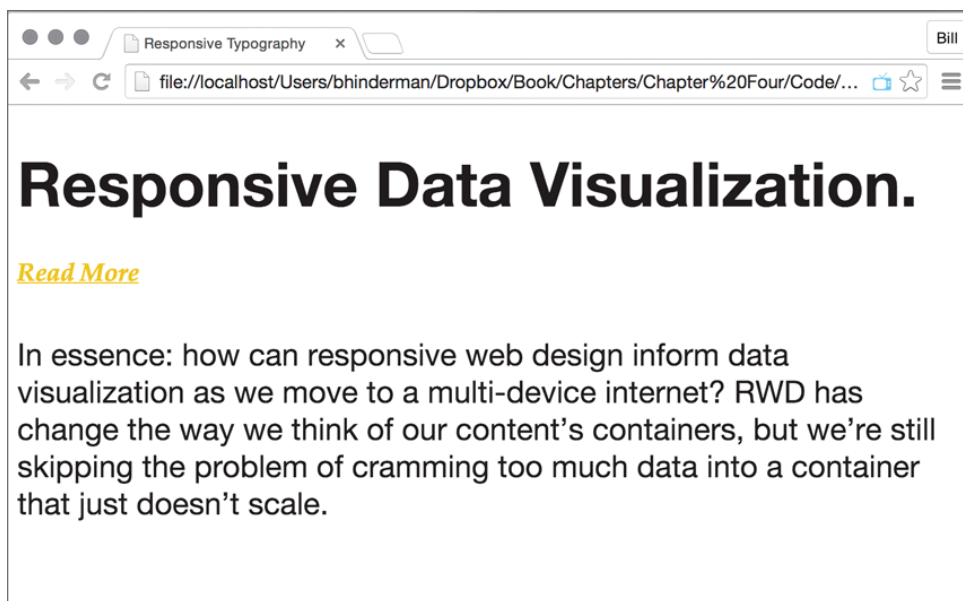


FIGURE 4-11

Your page on a tablet-size screen after changing the base font size to 1em

At this point, you've solved for large screens, by setting the base to 2 em; you've solved for medium screens, by setting it to 1.5 em; and you've solved for small screens by not setting it at all. However, now you have to choose which one you actually want to go with, because you can't use all three, right?

Wrong. Add the following code:

```
@media screen and (max-width: 768px) {  
  #container {  
    font-size: 1.5em;  
  }  
}  
@media screen and (max-width: 540px) {  
  #container {  
    font-size: 1em;  
  }  
}
```

With this bit of magic, you can use all three of these rules exactly when you want. Time for the big reveal.

Responsive Layout Design

You know that part in a bad action movie when the hero says the name of the movie and you really hope he's going to wink at the camera? This is that part.

You've done all that you can by being fluid and flexible. Your websites are container agnostic, and they are abstracted away from pixels, but there is still a point at which they don't work.

That's fine. You can start to *ask* browsers about their features, and then appropriately *respond* to those features.

The Media Query

You saw the CSS media query in action in the last piece of code, but let's get serious. Meet your new best friend, the media query. Ignoring everything else you did in the previous example, now focus on what you're actually trying to change:

```
#container {  
  font-size: 2em;  
}  
@media screen and (max-width: 768px) {  
  #container {  
    font-size: 1.5em;  
  }  
}  
@media screen and (max-width: 540px) {  
  #container {  
    font-size: 1em;  
  }  
}
```

Hold on, though. Wasn't the point of using 16px on the mobile screens to *not* have to write a rule at all? Here, you're overriding 2 em with 1.5 em, then overriding

1.5 em with 1 em. In a moment, you'll see how to solve this by writing your media queries mobile-first, and specifying a `min-width`, instead of `max-width`.

First, it's worth seeing just all the media query has to offer, more than just asking for screen width.

Media Types

Part of the CSS2 specification, way back in 1998, was a description of the *media type*. Per the W3C:

One of the most important features of style sheets is that they specify how a document is to be presented on different media: on the screen, on paper, with a speech synthesizer, with a Braille device, etc. Certain CSS properties are only designed for certain media (e.g., the “page-break-before” property only applies to paged media). On occasion, however, style sheets for different media types may share a property, but require different values for that property. For example, the “font-size” property is useful both for screen and print media. The two media types are different enough to require different values for the common property; a document will typically need a larger font on a computer screen than on paper. Therefore, it is necessary to express that a style sheet, or a section of a style sheet, applies to certain media types.

W3C MEDIA TYPES SPECIFICATION

The gist is this: Not all browsing styles lend themselves to the exact same presentation. Moreover, CSS separates the presentation layer from the content layer, so you can point entirely different media-targeted stylesheets to different types of media.

The most basic example, which is poorly described in the W3C spec, is screen versus printer. Well before the phrase responsive design was coined, web developers were writing separate stylesheets targeted toward print media. They were *responding* to the type of media. This could be done in two different ways. One way was in the actual markup, pulling in files per media like the following:

```
<link rel="stylesheet" href="screen.css" media="screen" />
<link rel="stylesheet" href="print.css" media="print" />
```

The other option was within the actual CSS, writing media-specific blocks of code:

```
@media screen {  
  #container {  
    font-size: 1.5em;  
  }  
}  
  
@media print {  
  #container {  
    font-size: 12pt;  
  }  
}
```

By default, CSS is applied to all media types (which is a specific media type, aptly named “all”); but when these rules are added, the browser assumes the responsibility of identifying itself within a class. If it’s a computer, it will be recognized as a screen, a printer, print, a screen-reader, speech. The underlying HTML doesn’t change, but the way in which it is presented to the user does.

The spec’s entire list of media types, although short, is surprisingly exhaustive:

- **All:** Suitable for all devices
- **Braille:** Intended for Braille tactile feedback devices
- **Embossed:** Intended for paged Braille printers
- **Handheld:** Intended for handheld devices (typically small screen, limited bandwidth)
- **Print:** Intended for paged material and for documents viewed on screen in print preview mode
- **Projection:** Intended for projected presentations—for example, projectors
- **Screen:** Intended primarily for color computer screens
- **Speech:** Intended for speech synthesizers
- **TTY:** Intended for media using a fixed-pitch character grid
- **TV:** Intended for television-type devices (low resolution, color, limited-scrollability screens, sound available)

Actually, Frankenstein Is the Doctor's Name

So what went wrong?

Well, when the spec was created, devices fit nicely into these buckets. Handheld, at the time, meant a lousy browser with virtually no presentation support and pitiful bandwidth.

As time passed, however, screens, device capability, and mobile browsers all got a lot better; and on the other end, TVs began to look more like computers. The “screen” media type began to creep into everything else that was *also a screen*.

Essentially, the types were not separated by form factor, but only by *current* capability gaps, and these gaps were filled in. A screen and a printer are not the same thing (at the time of writing), but a screen and a handheld are. Everything with a glass screen is a computer, the different names are just relics from a time when they weren't.

Padding in the Dark

Media types got developers part of the way there, but we just weren't getting enough information about users' browsers to accurately target the presentation we were aiming for. Therefore, the W3C set out to improve upon this with CSS3. The result was media queries, which take media types to the next level of granularity by not only identifying the *type* of screen, but also the *characteristics* of that screen.

Here's a portion of the code you wrote for the preceding exercise:

```
@media screen and (max-width: 768px) {  
  #container {  
    font-size: 1.5em;  
  }  
}
```

The media query in that code is in bold, and can be separated into two parts:

1. The media type, as per the CSS2.1 specification—in this case, **screen**
2. The media query, in parentheses—in this case, **(max-width: 768px)**

The query looks a lot like a CSS rule declaration, but it can be thought of in the exact opposite way. Remember that in a declaration the elements are property and value. When declared, you are *setting* the element’s property to that value.

In a media query, you are *testing* whether or not the property satisfies that value. In plain English, you are approaching the browser and asking, “Are you a screen? And if so, are you 768px wide or smaller?”

If the browser responds, “Yes, I am. And yes, I am!” then you enter the block within the media query and attempt to apply those CSS rules. In this case, the example shows the media query being used within your CSS; but just as media types can be used to import entire CSS files, so can media queries, as shown in this example:

```
<link rel="stylesheet" href="style.css" media="screen" />
<link rel="stylesheet" href="small.css" media="screen and (max-width:
768px)" />
```

The Web’s William Faulkner

At this point, you’ve seen a handful of scenarios in which you can definitely check for max-width. But it’s not an exhaustive rule set if it isn’t...well...exhaustive. Table 4-2 shows the exhaustive set of items that you can check for when querying a browser.

TABLE 4-2
CSS Level 3 Media Query Features

Feature	Description	Min/Max
Width	The width of the viewport display area of the output device	Y
Device-width	The width of the rendering surface of the output device	Y
Height	The height of the viewport display area of the output device	Y
Device-height	The height of the rendering surface of the output device	Y
Orientation	Either “portrait” or “landscape,” based upon the ratio of the height media feature to the width media feature	N

Feature	Description	Min/Max
Aspect-ratio	The ratio of the value of the width media feature to the value of the height media feature	Y
Device-aspect-ratio	The ratio of the value of the device-width media feature to the value of the device-height media feature	Y
Color	The number of bits per color component of the output device	Y
Color-index	The number of entries in the color lookup table of the output device	Y
Monochrome	The number of bits per pixel in a monochrome frame buffer	Y
Resolution	The density of the pixels in the output device	Y
Scan	The scanning process of “tv” output devices	N
Grid	Specifies whether the output device is grid or bitmap	N

Clearly, some are more useful than others. For the most part, you will be primarily focusing on width, height, and orientation in the examples in this chapter, but the logic can be trivially swapped out for other queries.

Browsers Are Liars

Unfortunately, browsers won’t always tell the truth about their features. One particularly egregious offender: The iPhone lies about its width. The iPhone 5, according to 100 percent of accounts, is 320px 980px wide. Hmm...

When the iPhone launched, there wasn’t a concept of responsive design yet. Apple figured that users would be fine pinching and zooming, so they added a new meta tag, the viewport. It does what you might expect, it lies to the rendering engine to create real estate that is “zoomed out” to fit 980px on a 320px screen. This is what websites that don’t have any responsive work added look like on a phone. It’s their desktop site, but really far away.

After Apple introduced the viewport tag, other mobile browsers adopted it, so although it isn’t a standard, it’s pretty much everywhere. Therefore, let’s bend it.

Rather than take the device's default (read: lies), you need to add this to your website's header:

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

This does two things. The `initial-scale=1` says that you're not zooming out; the scale of your website is 1:1 to the scale of the device's pixels. The `width=device-width` says that your website's width is not spoofed to 980px, but is in fact the viewport's *actual* width. The truth shall set you free. You can see this in Figure 4-12.

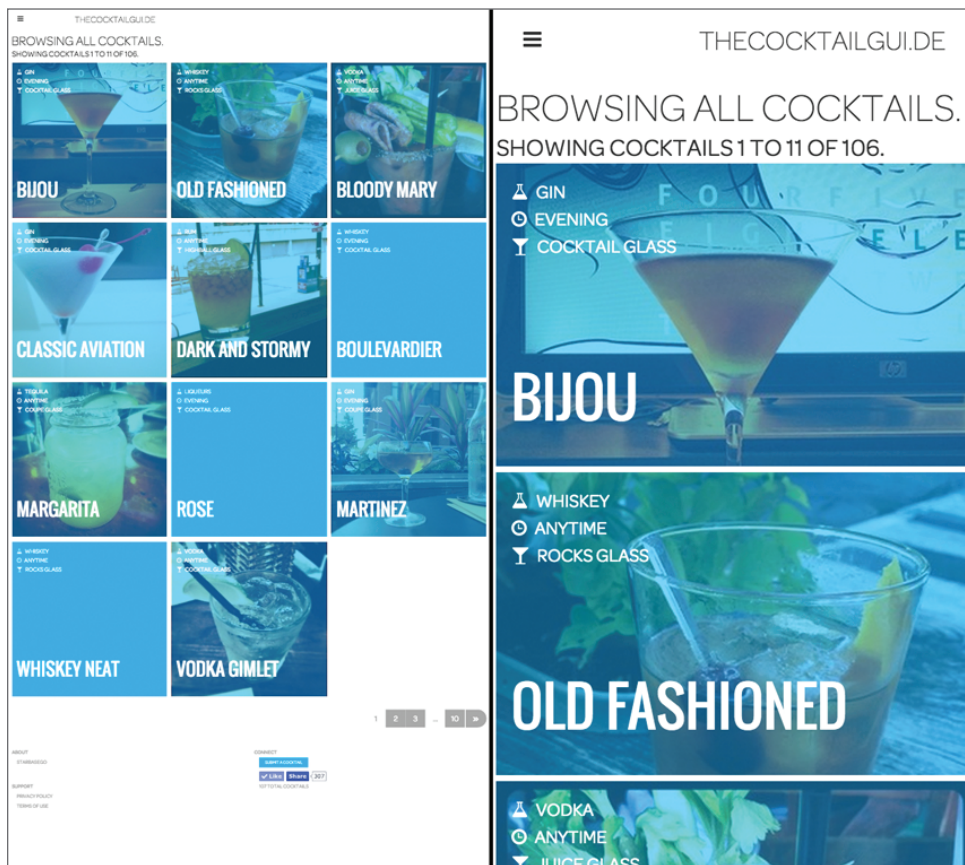


FIGURE 4-12

Browsers being liars. The left shows a website without the correct viewport meta tag, the right with it.

Fluid, Flexible, Responsive

Now that you have a deeper knowledge of what you were actually doing toward the beginning of this chapter, take one last look at the example, to ensure that you are doing everything you can to adapt your content in a lightweight way.

Fluid

```
#container {  
  margin: auto;  
  max-width: 1200px;  
  width: 100%;  
}
```

Check. Your entire site is within a fluid container that occupies as much room as it can, while still stopping at a reasonable width.

Flexible

```
#container {  
  font-size: 2em;  
}  
h1 {  
  font-size: 2em; /* 64px/32px = 2em */  
}  
h1 a {  
  color: #f1c40f;  
  font-family: "Book Antiqua", Palatino, "Palatino Linotype", "Palatino  
LT STD", Georgia, serif;  
  font-size: .40625em; /* 26px/64px = .40625 */  
  font-style: italic;  
}  
p {  
  /* No font size rule needed */  
  margin-top: .5em; /* 16px/32px = .5em */  
  padding-bottom: 1em; /* 32px/32px = 1em */  
}
```

Check. Not a single pixel in sight (in site). You've successfully abstracted all of the pixel values the designer originally wanted at the *ideal* size, while still letting them rely on the base container font-size.

Responsive

```
#container {  
  font-size: 2em;  
}  
@media screen and (max-width: 768px) {  
  #container {  
    font-size: 1.5em;  
  }  
}  
@media screen and (max-width: 540px) {  
  #container {  
    font-size: 1em;  
  }  
}
```

Check. As you began to scale your viewport size down, it became apparent that the base sizes just weren't cutting it, so you added media queries to check that you're on a 768px wide screen or smaller (`max-width: 768px` on line 4), and changed your base font size. Then, again at 540px wide, you shrunk the base font size further.

Because the 540px rule applies *after* the 768px rule in the CSS file, you know that it takes priority.

That would seem to cover all of our bases; but as you can see, there are a lot of pages left in this chapter.



NOTE

CSS priority rules are shown in Table 4-1.

Retrofit Responsive

Look again at the last rule. You're setting the font-size of your `#container` to 1 em. Isn't that the default? The point of all this is to make things look good on all devices, but should that really come at the cost of heavier code?

Answer: No.

What you did here was solve for the smaller cases by adding more CSS to your *desktop-focused* design. You were given a PSD of a 1200px `#container` on a screen size even bigger than that, and you increased your payload to fix the sizing as the screen size got smaller.

I like to call this a *retrofit responsive* technique. You didn't bake responsive practices into your initial design process, and as such you're stuck with a bunch of overrides in your CSS.

What if your relevant CSS rules looked like this instead:

```
#container {  
  /* No declaration, so default is 1em == 16px */  
}  
@media screen and (min-width: 541px) {  
  #container {  
    font-size: 1.5em;  
  }  
}  
@media screen and (min-width: 769px) {  
  #container {  
    font-size: 2em;  
  }  
}
```

That only manages to reduce a grand total of one line of CSS in this simplified case, but consider what it means for your overall design mindset. You started at the smallest screen you were planning to target, and as you *added* real estate, you *added* new rules.

Sounds awfully *progressive* to me...



NOTE

Remember when moving from max-width to min-width, you'll need to add one to your screen sizes (and subtract when moving from min-width to max-width).

Starting from the Bottom

You got a respite for a while, but remember that an important theme here is building mobile-first. As you were retrofitting responsive styles onto your previous designs, that full weight was being added to smaller screens, and that's not good.

All it takes is a mindset change. Design for mobile *before* designing for desktop, not after, and the CSS will follow.

Why Do We Start Small?

I've harped about it enough: Mobile users are the majority, but they're on weaker devices. Therefore, here's what you should consider when writing mobile-first responsive rules: The CSS is going to get bigger no matter what. That's a file that is downloaded, and you're stuck that way. What you can improve is the *rendering complexity* of elements. Consider the following two screenshots of developer tools on a screen that is approximately 320px.

In Figure 4-13, you took your desktop site and added rules to change the font-size as the screen got smaller. Notice that the default and 768px rules are applied, then immediately overwritten.

In Figure 4-14, you added min-width rules for larger screens. There is nothing to override because nothing else has been applied yet.

Styles	Computed	Event Listeners	DOM Breakpoints	»
element.style { }				+ [icon] [icon]
@media screen and (max-width: 540px) #container { <input checked="" type="checkbox"/> font-size: 1em; }				<u>simple.css:24</u> <u>simple.css:25</u> +
@media screen and (max-width: 768px) #container { font-size: 1.5em; }				<u>simple.css:19</u> <u>simple.css:20</u>
#container { font-size: 2em; }				<u>simple.css:1</u>
div { display: block; }				user agent stylesheet

FIGURE 4-13

Rules applied to an element on a mobile device using max-width overrides

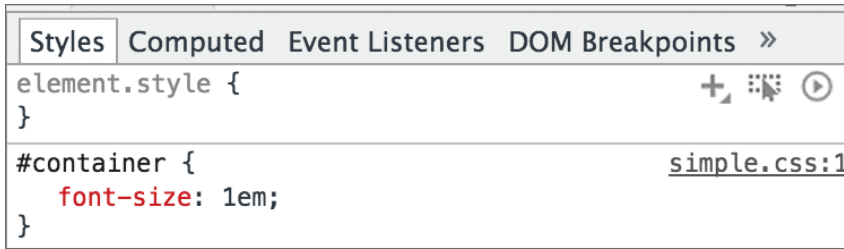


FIGURE 4-14

Rules applied to an element on a mobile device using min-width overrides

In addition, based on my experience, in just about every project this actually leads to lighter code than retrofitting. You're adding fewer rules to change things as the screen gets bigger, because you have more bandwidth and real-estate-considerate defaults.

How You Can Flip the Rules

When I say *flip* the rules, I mean move from a desktop-first pile of CSS with max-width and max-height queries to a mobile-first approach using only min-width queries.

Typically, it isn't all that hard, and it actually involves more deletion of code than addition. For example, suppose you have the following HTML for a basic navigation menu on a blog:

```
<div id="container">
  <ul id="nav">
    <li><a href="/">Home</a></li>
    <li><a href="/blog">Blog</a></li>
    <li><a href="/contact">Contact</a></li>
    <li><a href="/about">About</a></li>
  </ul>
</div>
```

The output is as exciting as ever (see Figure 4-15).

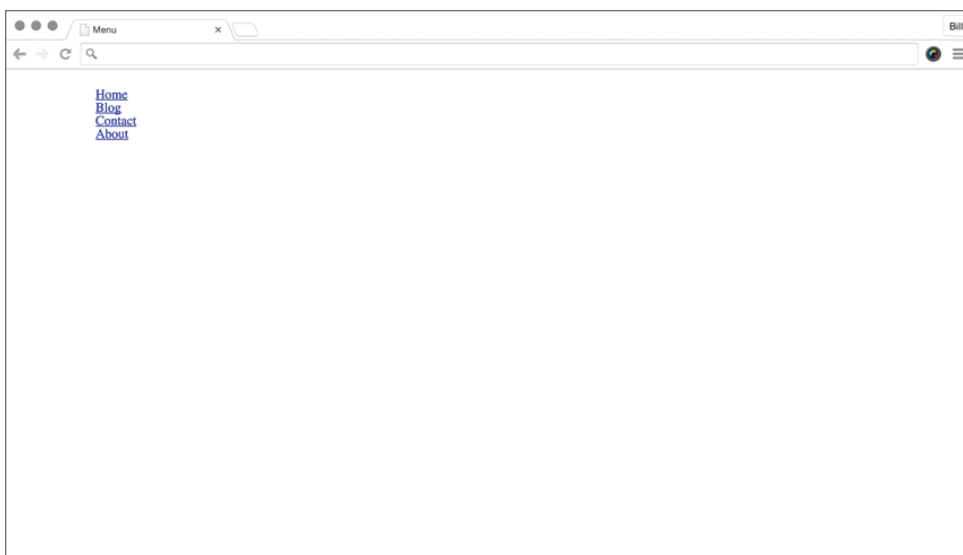


FIGURE 4-15

Your list, with no styles applied

The designer gives you the desktop-first PSD shown in Figure 4-16, and you write the following code; but rather than even look at it on the desktop, you test it on your mobile browser, and immediately see that it doesn't work (see Figure 4-17).

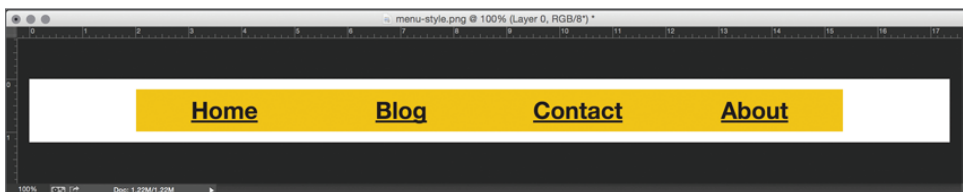


FIGURE 4-16

The designer's desired output for the list of links

```
#container {
  font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
  margin: 1em auto 0;
  max-width: 960px;
  width: 100%;
}
#nav li {
  background: #f1c40f;
```

```

display: block;
float: left;
font-size: 2em;
padding: 1.25%;
text-align: center;
width: 25%;
}
#nav li a {
  color: #231f20;
  font-weight: bold;
}

```



FIGURE 4-17

The list of links really doesn't work on mobile screens.

Let's first retrofit your responsive rules to make this layout work. If you just eyeball it, everything still *seems* fine if you throw the following rules on top:

```

@media screen and (max-width: 768px) {
  #nav li {
    width: 50%;
  }
}

@media screen and (max-width: 540px) {
  #nav li {
    width: 100%;
  }
}

```

But that doesn't really follow the vision of the designer. This is the problem so often with retrofitting responsive rules onto a design: They become a kind of best guess before you move on, and they can wreak havoc on an intended design.

Instead, try some math, and working mobile-first.

You know that the style that's breaking right now is the width of the `` elements. Therefore, rearrange your $\text{target} / \text{context} = \text{result}$ formula to extract your target value:

$$\text{result} \times \text{context} = \text{target}$$

Your *ideal* context is 960px, and your result is the width you've already attached to the ``: 25%. Therefore, your *ideal* target is 240px. That means that you need a *minimum* of 240px for every `` in order to have enough real estate. Change your default `#nav ` width to 100%, and assume your base case is a screen width of 240px (Figure 4-18):

```
...  
#nav li {  
  width: 100%;  
}  
...
```

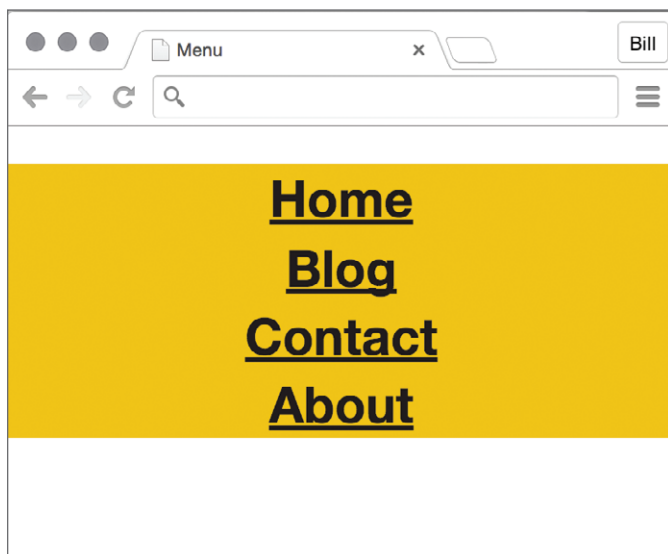


FIGURE 4-18

For mobile devices, default the links to take up 100% width.

Right now, the four `` elements are displayed in a row, and each takes up 240px, but what if you put them in a 2x2 grid instead? The minimum amount of width you would need is $240\text{px} \times 2 = 480\text{px}$, so set a rule there with the following code, which produces Figure 4-19:

```
...
@media screen and (min-width: 480px) {
  #nav li {
    width: 50%; /* 480px / 2 = 240px */
  }
}
...
```

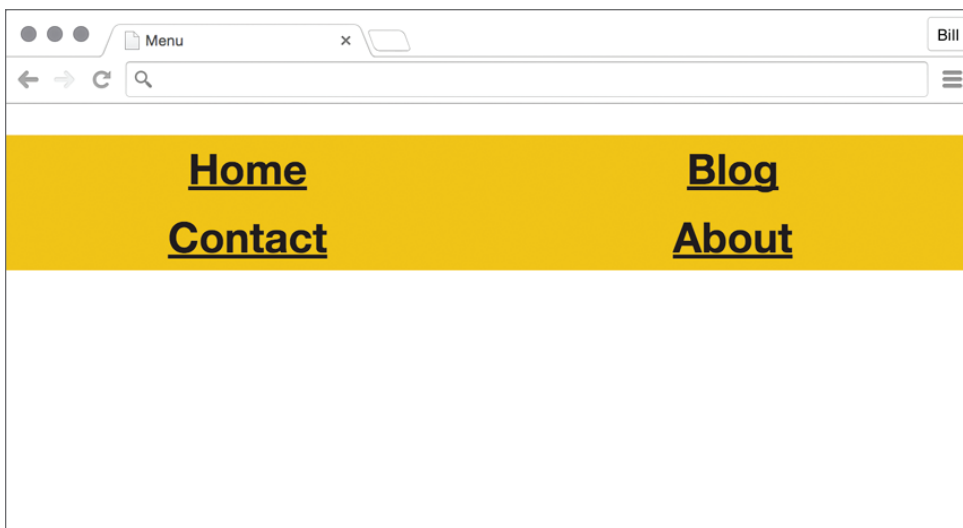


FIGURE 4-19

Once you have enough space, let the links take up 50%, floating next to one another in a 2x2 grid.

Now, the next logical cut would be back to four inline elements. $240\text{px} \times 4 = 960\text{px}$. What are the odds? The output is Figure 4-20.

```
...
@media screen and (min-width: 960px) {
  #nav li {
    width: 25%; /* 960px / 4 = 240px */
  }
}
...
```

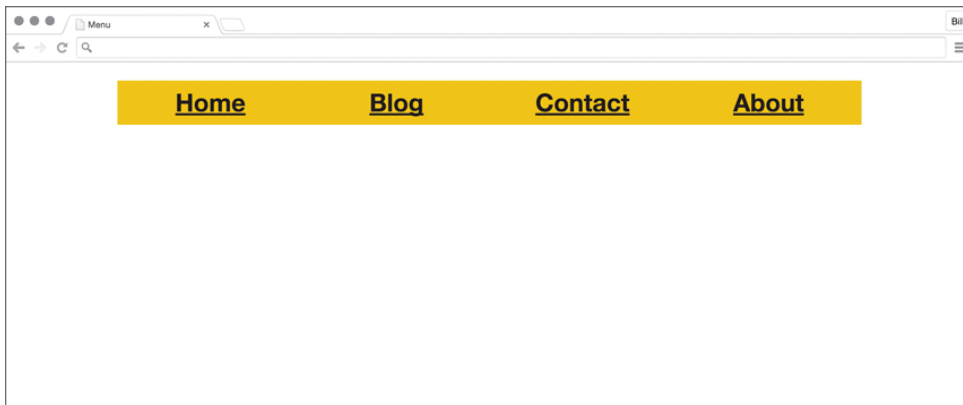


FIGURE 4-20

Finally, back at 960px, the links occupy 25%, again matching the designer's wishes.

Taken as a spectrum of screen sizes, the design flows as shown in Table 4-3.

TABLE 4-3

How the List Items Behave at Various Screen Widths

Width Range	li Behavior
0px–39px	Not considered
240px–479px	Full width 240px–479px
480px–959px	2-up 240px–479px
960px+	4-up 240px

Keep in mind that this follows the logical, progressive, enhancement-led concept of adding rules as you add real-estate. With that under your belt, let's move on to a more robust project.

The Cocktail Guide

No more simplified blogs. In this section you're going to build something useful. A few years ago, I had some friends over for a Christmas party and was mixing some fancy drinks. I have a few cocktail books in my kitchen—standard ones, ones that

have cute little made-up stories and dialogues about drinks, and one that's just plain good-looking. I always seem to gravitate toward that one, the good-looking one, when I need to double-check ratios or I'm in the mood to make something new.

On this occasion, I was making a Martinez, or an Aviation, or one of those drinks that I always make because it meets that important double requirement of tasting delicious and allowing me to look really cool making it.

So I cracked open the book, checked something about a vermouth ratio, and one of my friends asked, "Why don't you just look it up online?"

To which my immediate, unprocessed, uninformed response was, "There isn't a single mobile-friendly cocktail recipe website on the whole Internet."

Of course, that's a pretty hefty claim to level against the entirety of the Internet, so you're going to change that. I'm tasking you with building a mobile-first cocktail recipe website.

Now, full disclosure: This is a project that I personally worked on a few years back. It lives at www.thecocktailgui.de. You'll be moving through it with *radically different* code than what is live in production, because you're much better at this sort of thing now. However, it's a project that you will be working with *for the rest of this book*, because it spans both responsive design and data visualization topics, it's mobile first, and it wholeheartedly embraces the grid upon which it's built.

Building for the Smallest Screen

For this section, you're going to focus on the website's search results page, as it will have the most responsively moving parts.

This time you'll start by e-mailing the designer before anything is even started, indicating that you want to shrink that canvas down to something much more mobile. In return you get what is shown in Figure 4-21.

Not bad—you've successfully started on the most limited device size that you're willing to support.



FIGURE 4-21

The Cocktail Guide's list view on a mobile-size screen

You can abstract out the following: navigation, a header, and a results list. For now, ignore the code that surrounds an individual result, and code the high-level layout of the page as something like this:

```
<nav>
  ...
</nav>
<div id="container">
  <header>
    ...
  </header>
  <div id="content" class="search">
    <ul class="drink-list">
      ... <!-- Individual results -->
    </ul>
  </div>
</div>
```

On this device, everything appears to be in one long column, except for the menu, but assume that that hamburger menu is just an overlay toggle for now—you'll worry about it later. In that case, the layout CSS for your default scenario is as follows:

```
nav {  
    display: none;  
}  
#container {  
    /* Nothing */  
}  
header {  
    /* Nothing */  
}  
#content {  
    /* Nothing */  
}  
#content .drink-list {  
  
}  
footer {  
    /* Nothing */  
}
```

There are of course plenty of style rules for actual...style, but you're focusing on layout right now.

In terms of your individual result, assume your HTML looks something like this:

```
<li class="drink">  
  <h3 class="drink-name">...</h3>  
  <ul class="drink-attributes">  
    <li>...</li>  
    <li>...</li>  
    <li>...</li>  
  </ul>  
</li>
```

Again, looking only at layout CSS, you have something very basic:

```
.drink {  
    display: block;  
    margin: 1.25%;  
    width: 97.5%;  
}
```

Up and Out

After looking at your design on larger screens, you are amazed that nothing breaks but disappointed because the site is now just one big wide column. You remind the designer of the subject of the previous chapter (grids), and get what is shown in Figure 4-22 back, for tablet-size screens.

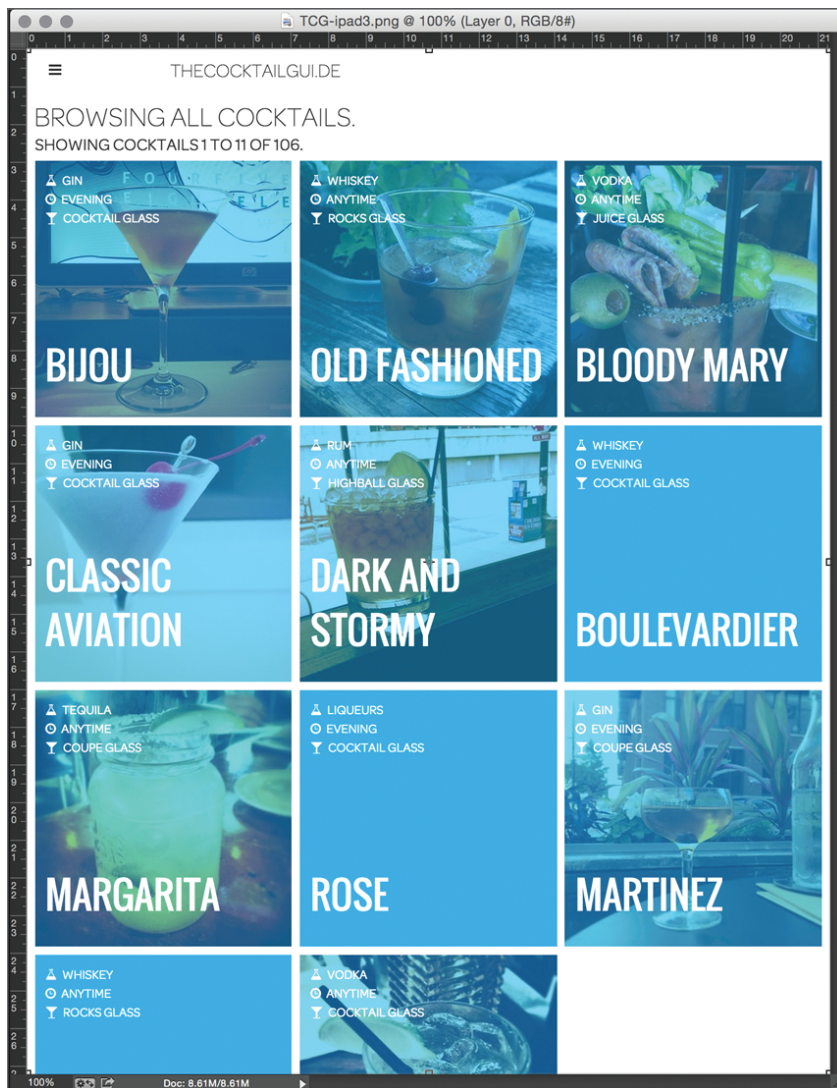


FIGURE 4-22

The Cocktail Guide's list view on a medium-size screen. The results now stack 3-up.

Other than on individual recipes in the list, it doesn't look like you need to make any changes. Add a media query. Your initial comp was 320px wide, but the designer specified that results need a minimum of only 240px to fit their content. Therefore, you can back into your media query by switching the target ÷ context = result formula. The designer gave you a design at 3-up, but do one better: Add a media query to put two results next to each other.

Here, 240px is your target width, and the result is a little complicated. If you had no gutters, it would be 50%, but you will have a grand total of three, each at 1.25%, so our result is as follows:

$$(100\% - (3 \times 1.25\%)) \div 2 = 48.125\%$$

That means the formula looks like this:

$$\text{Context} = 240 \div 0.48125$$

The context for the first media query is 498.701298701px...

Content Sets Size

That's gross. Although gross numbers were fine when you were setting values in your CSS, that doesn't apply as well here. There isn't a screen out there that is going to respond with "498.701298701px" when you request its width. You need to round; and because what you found was the *absolute minimum* pixel width for which your design will work, you need to round *up*.

Therefore, your first media query is `@media screen and (min-width: 499px)`. You might be asking yourself how many weird devices would be exactly 499px wide, as that's what you're targeting for. The answer is probably none, or maybe 50. It doesn't matter.

You are not solving for devices. The designer sent you a PSD named "cocktail-Guide_iPad3" and you ignored it because your *content* sets your sizes. Size-based media queries can and should be set at whatever pixel values make sense for the content, not for the handful of popular devices you might be targeting. You are solving for the entire river of devices flowing towards you.

Back to the size you just solved for, though, add your code:

```
.drink {
  display: block;
  margin: 1.25%;
  width: 97.5%; /* 100% - 1.25% - 1.25% */
}
@media screen and (min-width: 506px) {
  .drink {
    float: left; /* Get our results to display next to one another */
    width: 48.125%;
  }
}
```

In addition, just to ensure that your designer is happy, also figure out the 3-up ratio. Again: target = 240px, result = $(100\% - (4 \times 1.25\%)) \div 3 = 31.666666667\%$.

Context = $240 \div 0.3166666667$.

Your context is 757.894736834, so your media query is placed at 758px; and your code is even simpler than the previous media query, because all you need to change is the width:

```
.drink {
  display: block;
  margin: 1.25%;
  width: 97.5%; /* 100% - 1.25% - 1.25% */
}
@media screen and (min-width: 506px) {
  float: left; /* Get our results to display next to one another */
  width: 48.125%;
}
@media screen and (min-width: 758px) {
  .drink {
    width: 31.666666667%;
  }
}
```

At this point, you could just keep applying this formula to determine when you need to set breakpoints for 4, 5... n results in a row if the context for your #content never changes. Remember that right now all of your top-level parents are a perfect 100% width column; but then the designer sends along the vision of the site shown in Figure 4-23, for a desktop-size screen.

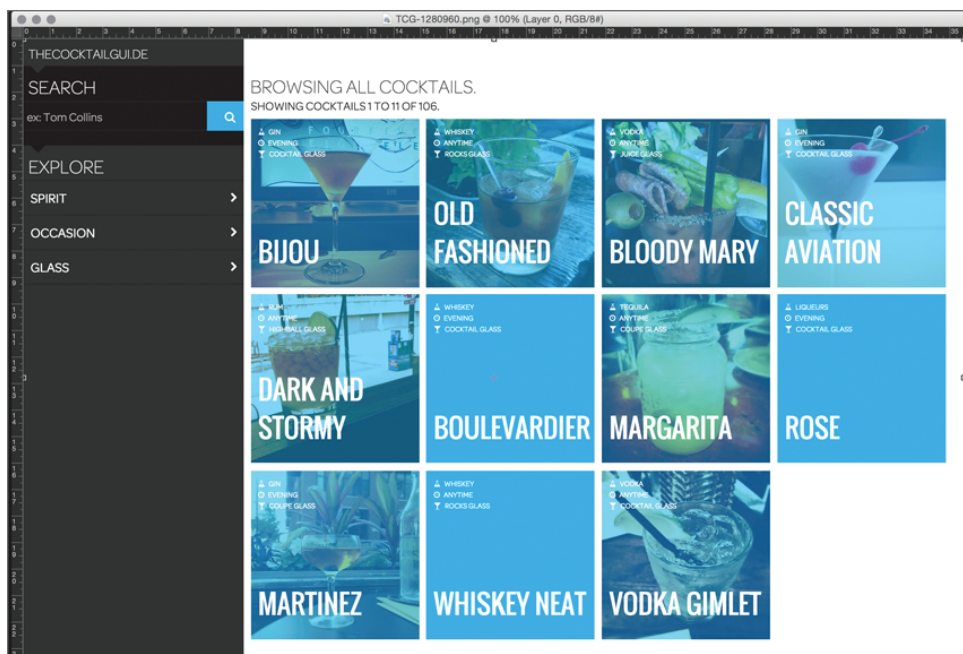


FIGURE 4-23

The Cocktail Guide's list on a desktop-size screen. Not only do the results now stack, but the menu has moved to a left column sidebar.

Your initial assumption about the menu has gone out the window, and now you have a section of the page taking up space *next to* the results list. That means that at a certain point, your results list's *result* in your formula is no longer 100%.

Classified Information

This poses a problem because of the way that media queries work. When you request a device's width, you are getting the width of the entire viewport, not the parent element you actually care about. In the exercise so far, those were identical, because the parent element was 100% of the page's width. Now, however, you're going to have to do some clever math to figure out where your breakpoint needs to be.

If you ask the designer and are told that the menu needs to be a minimum of 300 px wide, that's your menu's *target*. You don't yet have *context* for the menu, because you don't know where the media query is, so we can't get a *result*.

You can see from the PSD that once you open the menu, you also have four results in a row, and those must still be at least 240px wide. First figure out your results' relative size. The target is 240px. Context is $(100\% - (5 \times 1.25\%)) \div 4 = 23.4375\%$. Therefore, the formula is as follows:

$$\text{Context} = 240 \div .234375$$

That means the context is 1,024px. You can add a rule to your result's CSS:

```
.drink {
  display: block;
  margin: 1.25%;
  width: 97.5%; /* 100% - 1.25% - 1.25% */
}
@media screen and (min-width: 506px) {
  float: left; /* Get our results to display next to one another */
  width: 48.125%;
}
@media screen and (min-width: 758px) {
  .drink {
    width: 31.666666667%;
  }
}
@media screen and (min-width: 1024px) {
  .drink {
    width: 23.4375%;
  }
}
```

That context, however, is in this case the width of the results list, not the entire page. You're not done yet. The page width is currently made up of `nav` + `#content`, giving you the following:

$$300\text{px} + 1,024\text{px} = 1,324\text{px}$$

The entire page's context is now 1,324px. From here, it's trivial to apply your original target \div context = result formula to the two top-level containers:

$$\text{nav: } 300\text{px} \div 1,324\text{px} = 22.6586103\%$$

$$\text{\#content} = 1,024\text{px} \div 1,324\text{px} = 77.3413897\%$$

Therefore, to wrap this media query up, you must also add the following rules to the top-level containers' CSS:

```
@media screen and (min-width: 1324px) {  
  nav,  
  #content {  
    float: left;  
  }  
  nav {  
    width: 22.6586103%;  
  }  
  #content {  
    width: 77.3413897%;  
  }  
}
```

Something was skipped, though. Your media query for the individual result happens at 1,024px, but your rule for putting the `<nav>` next to the `#content` doesn't happen until you're 1,324px wide. A lot can happen in those 300px.

I'll skip the math, but you actually should be increasing to 5-up when your context hits 1,298px. It's persnickety, but you need to add two more rules to your results:

```
.drink {  
  display: block;  
  margin: 1.25%;  
  width: 97.5%; /* 100% - 1.25% - 1.25% */  
}  
@media screen and (min-width: 506px) {  
  float: left; /* Get our results to display next to one another */  
  width: 48.125%;  
}  
@media screen and (min-width: 758px) {  
  .drink {  
    width: 31.666666667%;  
  }  
}  
@media screen and (min-width: 1024px) {  
  .drink {  
    width: 23.4375%;  
  }  
}  
@media screen and (min-width: 1298px) { /* Up to five */  
  .drink {
```

(continues)

(continued)

```
    width: 18.5%;  
  }  
}  
@media screen and (min-width: 1324px) { /* Back down to four */  
  .drink {  
    width: 23.4375%;  
  }  
}
```

At 1,298px width, you up your count to five, but when your context changes, you reduce it back down to four. We'll stop this exercise here, before heads explode.

The Reusable Responsive Grid

That was a lot of work, but sometimes projects take that sort of long, tedious math to figure out *perfect* increments. Most of the time, however, especially when prototyping, you can find some corners to reduce that effort that will get you *close enough* before you ever have to break out the calculator.

In Chapter 3, you walked through the logic of creating a usable grid. Remember the five components?

- **Unit:** The base module for a grid
- **Gutter:** The spacing between each unit
- **X-count:** The number of units in a row
- **Column:** The vertical slices of units
- **Row:** The horizontal slices of units

Now you're going to actually make one in HTML and CSS.

Building a Reusable Grid

As much as I have talked about not ever letting one level of your progressive enhancement tower reach *down* and break another, there are certain points where lower levels can assist you by reaching *up*. In this section, you will add classes to your HTML elements that don't help your semantics in any way but do help you conform to a grid.

I want to make this very clear: You are, in one way, breaking the semantic cleanliness of your markup. This is a tradeoff for the ease of implementing a reusable grid. All of this work can be done within your semantic markup—it will simply take longer.

First look at a grid's markup:

```
<div class="row row-12">
  <div class="col-4"></div>
  <div class="col-4"></div>
  <div class="col-4"></div>
</div> <!-- end row -->
<div class="row row-12">
  <div class="col-3"></div>
  <div class="col-3"></div>
  <div class="col-6"></div>
</div> <!-- end row -->
<div class="row row-12">
  <div class="col-2"></div>
  <div class="col-2"></div>
  <div class="col-5"></div>
  <div class="col-3"></div>
</div> <!-- end row -->
```

It doesn't mean much yet, but maybe you can see where we're going. Layer on the CSS that makes it do something:

```
.row {
  display: block;
  position: relative;

  &:before, &:after {
    content: "";
    display: table;
  }
```

(continues)

(continued)

```
&:after {
  clear: both;
}
}
.row > [class^="col-"],
.row > [class*=" col-"] {
  display: block;
  float: left;
  position: relative;
  width: 100%;
}
```

You still don't have actual flexible columns yet, but this code tells you that when you're inside a `.row` and immediately find a `.col-`, you need to float each of them left, and box their positioning.

Now suppose you're working with a 12-column grid and have decided (for math's sake) to completely remove your gutter. Instead of manually writing out each row and column's rules, you can use a CSS extension language to do it for you. CSS extensions like Sass or Less allow you to add variables, nested rules, mixins, inline imports, and more to your CSS.

In the code below, you see Sass (Syntactically Awesome Stylesheets) in use to quickly loop through each column in a row:

```
.row-12 {
  @for $col from 1 through 12 {
    .col-#{ $col } {
      width: percentage($col/12)
    }
  }
}
```

This is simply shorthand to say that this element's width is equal to the percentage modifier of its column value divided by 12. In essence, it's doing exactly what you were doing when you computed your Cocktail Guide, just on a grid. Without gutters, it means that a `.col-3` would have the width $3 \div 12 = 25\%$.



NOTE

Sass and Less have similar feature sets. Either one can massively speed up your development time. Learn more about Sass at <http://sass-lang.com/> and Less at <http://lesscss.org/>.

Building a Responsive Grid

The good news is that you're already almost there. You've built a grid that's totally flexible and can be applied agnostically to the project's design.

The difference now is picking your breakpoints—that is, the points at which you reset your sizes. In the Cocktail Guide exercise, you took the long, dumb, stumbling way to figure out exact sizes that worked for your content. When building your reusable grid, you can pick sizes now and tweak them later.

Adding Abstracted Sizes

Again assume you're using a 12-column grid; but this time you'll bucket your sizes. For the grid, let's say you want to make responsive decisions for small at 320px, medium at 760px, and large at 920px. The names will make sense in a moment. Your markup for *one row* worth of columns could look like this:

```
<div class="row row-12">
  <div class="col-s-12 col-m-6 col-l-7"></div>
  <div class="col-s-12 col-m-4 col-l-3"></div>
  <div class="col-s-12 col-m-2 col-l-2"></div>
</div> <!-- end row -->
```

See what you're doing? Now add the CSS:

```
.row-12 {
  @media screen and (min-width: 320px) and (max-width: 759px) {
    @for $col from 1 through 12 {
      .col-s-#{ $col } {
        width: percentage($col/12)
      }
    }
  }
  @media screen and (min-width: 760px) and (max-width: 919px) {
    @for $col from 1 through 12 {
      .col-m-#{ $col } {
        width: percentage($col/12)
      }
    }
  }
  @media screen and (min-width: 920px) {
    @for $col from 1 through 12 {
```

(continues)

(continued)

```
.col-1-#{ $col } {  
  width: percentage( $col/12 )  
}  
}  
}
```

All you've done is size-box your grid. You now have uniform breakpoints for small, medium, and large at which you can change the targets for every element within the grid at the same time.

As with the unresponsive reusable grid, it's a powerful prototyping tool, but it can lead to bloated markup. Use it at your own risk.

Fortunately, many CSS frameworks already have these grids baked in. Twitter Bootstrap and Zurb Foundation have powerful grids that work similarly, and they can be used out of the box.

I'll say it once more, though: Be wary. Your progressive enhancement tower is shifting beneath your feet.

Summary

You now have every tool you need to build a responsive website without JavaScript. You've learned the rules around how CSS works, and what levels of precedence each type of styling takes. You've also learned how to use the media query to start targeting your CSS to specific sizes of screen.

You have seen how to start retrofitting responsive CSS rules onto a non-responsive website. Hopefully you've decided that building your CSS media queries using min-prefixed screen size rules is a smart way to push a mobile-first mindset through your code.

Even more than that, you've learned how to use this CSS to build a reusable layout grid for your projects.

Try It

Go forth and build rectangles. You now have the tools to build fluid, flexible, responsive layouts, so put those tools into action.

- **Easy:** Go back to Chapter 3's final product and add some responsive break-points where content begins to break. There should be (at least) two.
- **Intermediate:** Brush up on your CSS by building out the results list from the Cocktail Guide. More detailed designs can be found at www.responsivedatavisualization.com/cocktail.
- **Hard:** Download the sample markup for the Cocktail Guide and convert it to use a 12-column grid. Get stuck? Maybe it isn't actually possible...

05

05

Enhancing with JavaScript

As we reach the last layer of the progressive enhancement tower, it's important to remember how far you can get without reading this chapter. Fluid grids, media queries, and flexible images are the core of responsive web design. JavaScript is how you will bridge the gap from web design to complex responsive data visualization.

Using JavaScript

Your website is a building. The markup is the rebar and cement. CSS is the drywall and furniture. JavaScript is the electricity. Electricity is amazing and important and makes a whole lot work, but if you're not careful, you can start a big fire with it.

JavaScript is a dynamic programming language. That means it executes its behaviors when the code is being *executed* on the client (runtime), rather than when it is being compiled.

In a static programming language, such as Java or Objective-C, the type of a variable must be declared explicitly in the code, like this:

```
String myString = "Hello world";
```

In a dynamic programming language, such as JavaScript, the variable does not need a type, as the language will figure it out at runtime. Here is an example:

```
var myString = 'Hello world';
```

Moreover, in a static programming language, once a variable is set as a type, it remains that type. Trying to do the following will result in a compile type warning:

```
String myString = "Hello world";  
myString = 1; // Compile type warning
```

In a dynamic programming language, you are free to change an object's type for the same variable. Names are bound to an object with an assignment operator. Therefore, the following works totally fine:

```
var myString = 'Hello world';  
myString = 1; // Go for it, superstar.
```

There are many more differences between these programming languages than just how type handling works, but the lesson here is that there's more freedom, and freedom to make mistakes, using a dynamic language.

Although JavaScript is available in plenty of non-web platforms, such as PDFs, desktop widgets, and virtual machines, for the purposes of this discussion it's

important to understand that JavaScript is part of the user's browser. It's used to implement scripts that interact with the user, modify the DOM, communicate asynchronously with the server, and control the browser itself.

JavaScript Enables Rich Interactivity

Simply put, the Internet would look a bit more boring without JavaScript.

Take, for example, Google's instant search introduced a few years back. When you access google.com (with JavaScript enabled), you see their classic white home page with a search bar. However, the moment you begin typing, your view is radically switched to what looks like a results list, as shown in Figure 5-1.

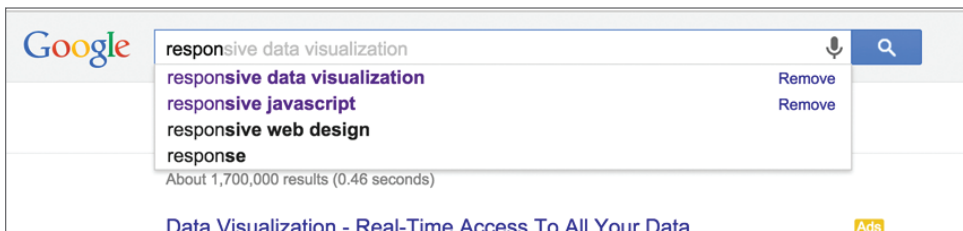


FIGURE 5-1

Google's instant search results in action

As you can see, after typing less than half a word of the search terms, Google is already displaying a ton of information. There hasn't been any page refresh; but as you type, Google polls the partial search query you are inputting and does the following:

- Checks saved searches that you have done before
- Matches popular searches
- Returns results for the partial search you are typing
- Displays sponsored listings and ads for the search you are typing

All of these behaviors are dynamically updated in your browser as you type, and all, seemingly, in an instant. If you perform the same search without JavaScript enabled, none of the instant search capabilities are available, as shown in Figure 5-2.

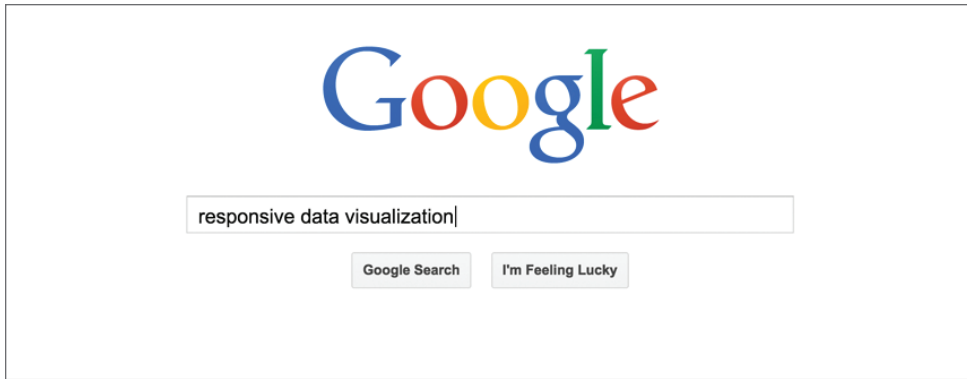


FIGURE 5-2

Doing the same search without JavaScript enabled removes all the instant search functionality.

With JavaScript disabled, the flow for executing a search is as follows:

1. Access google.com.
2. Input your full search.
3. Submit the form to execute the search.
4. Wait.
5. Reach a brand new page with your results.

The result is exactly the same, but the interaction just somehow *feels* worse, right? Doesn't it feel dated?

With JavaScript still disabled, try now to check your Facebook news feed. You're greeted with the message shown in Figure 5-3. Facebook.com does not work *at all* without JavaScript.

We're spoiled by the amount of rich, interactive web applications that now either rely heavily on, or are completely broken without, JavaScript running on the client.

That sounds like it's breaking a core tenet stressed in this book so far, doesn't it?

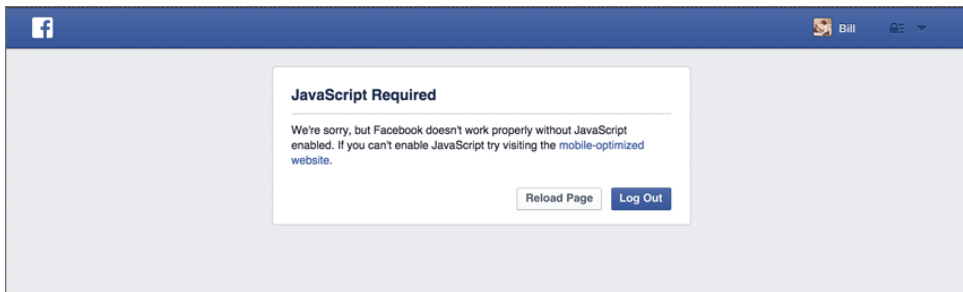


FIGURE 5-3

Facebook without JavaScript enabled is entirely unusable.

Remember Progressive Enhancement

In the progressive enhancement tower previously described, JavaScript sits at the very top. Although it is basically ubiquitous in good *modern* browsers, you shouldn't assume that your users will be accessing your content through one of those. Moreover, as shown in Table 5-1, it fails much more visibly than CSS or HTML, and it is simply quirky from browser to browser.

TABLE 5-1

How HTML, CSS, and JavaScript Handle Failure

Language	Failure Type	Failure Handling
HTML	Unknown tag	Treat as block element.
CSS	Unknown	Ignore rule.
JavaScript	Literally anything	Show error in browser console; stop parsing any JavaScript that comes after it.

Mobile-first building means careful consideration of just how much JavaScript you're planning to use, and how you're going to use it to alter the rendering and behavior of your website on various device types.

There are three high-level ways to think about how you're going to include JavaScript in your website:

JS-Adaptive

Straightforwardly, with this method, all devices are served identical HTML, CSS, and JavaScript content. Only when the JavaScript is executed on the device does the site's behavior change.

It's worth noting that this is the method that Google recommends using. Note also that it is, by far, the least amount of work.

Dynamically Served

With this method, all devices get the same HTML, but the JavaScript is served from a URL that dynamically sends the browser *different* JavaScript based upon the device's user agent.

Using this configuration, all devices get the same HTML; and in that HTML, the `<script>` element reaches an external JavaScript file that can contain different content based upon the requesting user agent.

The problem with this approach, as you may already have gathered, is that this forces you to write different JavaScript based upon these different device user agents, meaning you are no longer creating abstract solutions for *device features*, and are instead solving for *individual devices*.

Combined Detection

This method relies both on JavaScript and server-side device capability detection to serve different content to different devices.

Combined detection relies on both the server producing dynamic JavaScript files and the client-side JavaScript detecting the device's capacities and altering the content being served.

Not only is different JavaScript sent back, but also the HTML and CSS can vary greatly per client, as the website can detect screen dimensions, user agent, features, and so on.

This is commonly referred to as RESS: responsive web design with server-side components; and while this can produce extremely targeted experiences, it is much more work than the JS-Adaptive method, and it breaks the core concept of web URLs being the *universal* way to reach *uniform* content.

Sensible Defaults

Don't worry. I'm not filling a chapter about JavaScript with excuses not to use JavaScript. The point is to get first to a working state without it. Then, get as close as you can to the ideal state without it. Then, write your JavaScript to make something working into something amazing. Consider the cocktail guide website from the previous chapter. If you recall, we sort of hand-waved around the way the menu toggle in the top-left corner worked (see Figure 5-4).

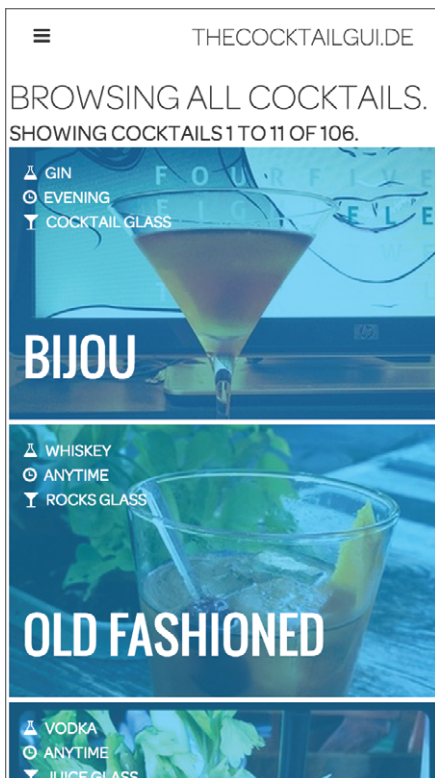


FIGURE 5-4

The menu toggle on the top left of the Cocktail Guide

Assume the menu has a bunch of categories, a search form, and a link to the home page. That's obviously too much navigation to try to fit above the actual content of your search results page. First solve it without any JavaScript:

```
<a class="menu-toggle" title="Menu" href="#menu">
  <i class="icon-menu"></i>
</a>
<div id="container">
  <header>
    <a title="Menu" href="#menu"><i class="icon-menu"></i></a>
  </header>
  <div id="content" class="search"></div>
  <footer></footer>
</div>
<nav id="menu"></nav>
```

A pattern that's been in use since the beginning of responsive design (or even simply mobile thought in web design) is the footer-anchor menu. That hamburger menu link in Figure 5-4 is simply an anchor link to the menu, as shown in the preceding code. The menu itself takes up the full screen width, and sits below the page's footer, as shown in the following code. The menu is shown in Figure 5-5.

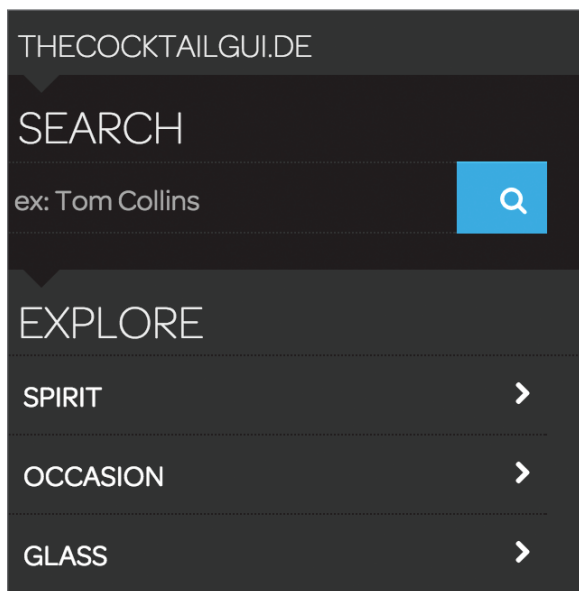


FIGURE 5-5

The Cocktail Guide's menu

If you were to view the website on a very long, narrow screen, it would look a bit odd, as the menu would be sitting below all of the page's content. That being said, this solution is essentially bulletproof. It does not *rely* on JavaScript, or even CSS (other than to make it visually attractive). The menu is there, but it's just out of the viewport until someone looks for it.

Now that you have your working case, you can begin to make things better. Instead of linking to the bottom, you want to push your menu off of the page's canvas to the left, and have it pull in next to the content when the menu button is clicked (see Figure 5-6).

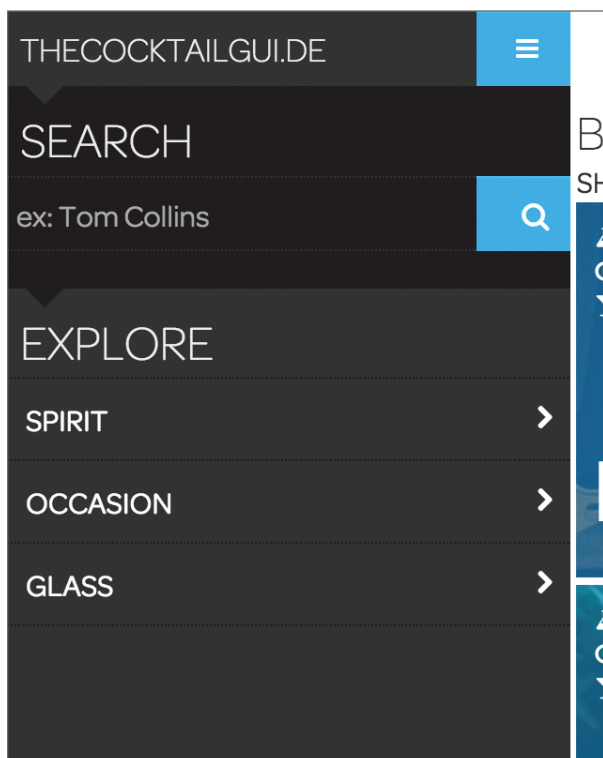


FIGURE 5-6

The menu attached to the left of the browser window

Doing that means you'll need to be able to toggle some behavior on the client when the user clicks the menu button; this sounds like interaction. You're now at a point that you need JavaScript.

However, if you were to first add the CSS to position your menu off-screen by default and users didn't have JavaScript working correctly, they would have no way to access the menu. That's no good. Therefore, instead, keep your default, and only change the way the page is laid out if you *know* that the user has JavaScript enabled. You can do this by traversing down your progressive enhancement tower, adding a CSS class *using* JavaScript.

Put more simply, if your design layer needs to change to allow for your interaction layer, ensure that the interaction layer is making that change. You'll use some jQuery here to simplify the code that does this:

```
$(function() {  
    $('body').addClass('js-enabled');  
});
```

Now, in your CSS you can change the way `#container` and `#menu` are positioned when the `body` tag has proven that JavaScript is running on the client:

```
/* Default styles above */  
.js-enabled #menu {  
    left: -320px;  
    position: absolute;  
    top: 0;  
    width: 320px;  
}
```

You're partway there. You now have a menu that is hidden off-screen when it isn't in use. Notice that it's moved -320px outside of the screen boundaries, and is 320px wide. Although it's invisible to the user, it is technically only 1px away from showing on the left side of the browser window. Suppose your menu has the same menu toggle (link) as the header, with the same class of `.menu-toggle`. You can now use JavaScript to conditionally add a class to your `#menu` to indicate that it should be visible:

```
$('.menu-toggle').on('click', function(e) {  
    e.preventDefault();  
    $('#menu').toggleClass('menu-open');  
});
```

The `toggleClass` function will add or remove the class `'menu-open'` from the `#menu nav` container whenever `.menu-toggle` is clicked. It's then trivial to add your CSS to position the nav on-screen:

```
/* Default styles above */
.js-enabled nav {
  left: -320px;
  position: absolute;
  top: 0;
  transition: left .3s; //Add a simple transition
  width: 320px;
}
.menu-open nav.menu-open {
  left: 0;
}
```

Notice above that there's a transition on the menu. This is a nice touch to make it feel like the menu is moving onto and off of the screen, rather than simply disappearing.

You now have a menu that *feels* more interactive, but keep going. If you recall from the same example, once you have enough room to keep both the menu and the search results next to one another on the screen, you want them both to always show. This is illustrated in Figure 5-7.



FIGURE 5-7

Both the menu and results are shown next to one another when there is enough screen space.

It doesn't seem like you're adding any more interaction with this change, though. It seems strictly like design. In fact, you solved for this in the previous chapter about using CSS, before knowing anything about JavaScript.

You can walk through that example again or just review the following snippet, with the small addition to hide the navigation toggle buttons at this size:

```
@media screen and (min-width: 1324px) {  
  nav,  
  #content {  
    float: left;  
  }  
  nav {  
    width: 22.6586103%;  
  }  
  #content {  
    width: 77.3413897%;  
  }  
  .nav-toggle {  
    display: none;  
  }  
}
```

It would be great if this just worked but you have added some conflicting specificity in the CSS selectors, by referencing the `.js-enabled` and `.menu-open` classes. However, because this is a design-level problem, you should find a design-level solution. That solution is using the `max-width` media query to limit your changes to the screen sizes for which they need to work. Here is the full simplified CSS:

```
@media screen and (max-width: 1323px) {  
  .js-enabled nav {  
    left: -320px;  
    position: absolute;  
    top: 0;  
    transition: left .3s;  
    width: 320px;  
  }  
  .menu-open nav.menu-open {  
    left: 0;  
  }  
}  
@media screen and (min-width: 1324px) {  
  nav,  
  #content {
```

```
    float: left;
  }
  nav {
    width: 22.6586103%;
  }
  #content {
    width: 77.3413897%;
  }
  .nav-toggle {
    display: none;
  }
}
```

You have now *progressively* built a navigation module that works perfectly without JavaScript, uses JavaScript nicely on screens for which it is suitable, and attains an ideal state on large enough screens. You're done, right?

Well, no. Keep in mind that this JavaScript is always going to be running in the background listening for clicks on the `.menu-toggle`, even when it is no longer clickable. What you need is a way to conditionally query the browser for features in JavaScript, just as you did in CSS.

Responsive JavaScript

That's exactly what you're going to do. If the preceding solution was an answer for *unobtrusive* JavaScript, you want to find a way to create lightweight *responsive* JavaScript that is only requested and used when the device's features allow for it.

Your first thought might be to run to polling for user agent strings and building for specific devices. Wrong. Bad. No. You're better than this. You've already seen that building for



NOTE

User agent polling is discussed as a helpful technique for RESS (and not much else) in Chapter 2.

one device doesn't help you in the long-term, as you'll then need to build separately for *every* device.

Server-Side Feature Detection

This gets you partway to where you want to be. Feature detection is the self-conscious, older brother of the media query that turned to creatine and protein shakes after he didn't make varsity.

It was briefly mentioned earlier, in the discussion about RESS and combined device detection for JavaScript loading. Essentially, this forces the work a media query *would* be doing onto the server. At its core, server-side feature detection still relies on decoding the browser's user agent string and then abstracting that out to different device functionalities. Libraries, expensive ones, must be kept up to date, and polled by your server-side code to check whether device X has a certain screen size, can be portrait/landscape, has retina resolution, and so on. Suppose you're querying simply for screen width. Your server-side code would need to create *different* files that all resolve to `"/script.js"` in the following code; and the file would need to be served with a `"vary:user-agent"` HTTP header:

```
<script type="text/javascript" src="/script.js"></script>
```

This totally solves the problem, but it forces you to create *different* assets (HTML, CSS and JavaScript) for different device types. The boundary here blurs between adapting to devices and simply creating new mobile websites.

Building from CSS

I've been poking holes in the other methods—it should be clear that a bridge between CSS and JavaScript is what we're looking for. The media query has helped tremendously in making responsive CSS; it needs to have a cousin in the JavaScript world.

Without actually relying on anything new, you can create a facsimile of a media query to poll your JavaScript. It works by pushing some content into the `:after` pseudo-class of your page, as shown in the following code:

```
body:after {  
  content: 'small';  
  display: none; /* Hide content */  
}
```

Here, on the page's `body` tag, you have added a piece of hidden content to indicate (building mobile-first as your default) which device you consider small, medium, and large.

You create your CSS media queries as normal, as shown in the following code:

```
body:after {  
  content: 'small';  
  display: none; /* Hide content */  
}  
  
@media screen and (min-width: 600px) {  
  body:after {  
    content: 'medium';  
  }  
}  
  
@media screen and (min-width: 960px) {  
  body:after {  
    content: 'large';  
  }  
}
```

The assumption here is that 600px and 960px are sizes at which you have decided a number of meaningful design changes need to happen; and as such you have been able to bucket these as *medium* and *large* device classes.

At this point, you have simply created a piece of content that sits in your DOM and automatically updates when the viewport width changes. What you need to add is a meaningful connection to your JavaScript.

It just so happens that vanilla JavaScript can poll the `:after` pseudo-class's content, as shown in the following code:

```
var query = window.getComputedStyle(document.body, ':after')  
              .getPropertyValue('content');
```

You now have a JavaScript variable which is populated with your CSS media query's above-set small, medium, or large value. Whenever the above code is run, it will look for the `:after` pseudo-class of the body, and get its content, which is set in CSS.

In your JavaScript, you can now *conditionally* load different JavaScript for small, medium, and large screens, as shown here:

```
var query = window.getComputedStyle(document.body, ':after')
                    .getPropertyValue('content');
if (query.indexOf('small') > -1) {
    //load small JS
}
if (query.indexOf('medium') > -1) {
    //load medium JS
}
if (query.indexOf('large') > -1) {
    //load large JS
}
```

This is a viable solution, and it makes sense that your JavaScript breakpoints should build upon, and reflect, your CSS breakpoints. However, it seems like you're taking a step in the wrong direction, cluttering up your CSS (and technically your DOM) with that `body:after` content for your JavaScript.

Additionally, unless you wrap this code in a check for window resize, it will only be accurate the first time it runs. If someone, for example, switches from portrait to landscape, and crosses the threshold of medium and large, the JavaScript won't be updated accordingly.

You *can* wrap the code in that resize check, but polling the DOM is an incredibly expensive thing to do in JavaScript, and you don't want to force it unnecessarily.

JavaScript's `matchMedia` Object

Enter the JavaScript-only solution to the media query: the `matchMedia` object.

The `matchMedia` object is a property of the JavaScript `Window` object, which represents the browser's open window. When queried, the `matchMedia` returns a

`MediaQueryList` object representing the parsed results of the window's media query string. Yes, the same media query string described in Chapter 4.

Here's what it looks like:

```
var mql = window.matchMedia(mediaQueryString);
```

In the preceding example, `mediaQueryString` would look something like `"(min-width: 600px)"`. It's exactly what you would expect.

The method to query whether or not the `matchMedia(mediaQueryString)` resolves to true is `.matches`. Therefore, the following reproduces what you earlier tried to base on CSS:

```
if (window.matchMedia('(max-width: 599px)') {  
  //load small JS  
} else if (window.matchMedia('(min-width: 600px) and (max-width:  
959px)').matches) {  
  //Load medium JS  
} else {  
  //load large JS  
}
```

The huge benefit you get from the `MediaQueryList` is that you can also add *listeners* for changes on it. Suppose, again, you were concerned about a user switching from portrait to landscape. You can add the following listener to poll only when the orientation changes:

```
var mql = window.matchMedia('(orientation: portrait)');  
mql.addListener(handleOrientationChange);  
handleOrientationChange(mql);  
  
function handleOrientationChange(mql) {  
  if (mql.matches) {  
    //The device is currently in portrait orientation  
  } else {  
    //The device is currently in landscape orientation  
  }  
}
```

You now have a 100% JavaScript solution that uses the same media queries you are already familiar with.

A note on browser support: It's not stellar. The `matchMedia` object is relatively young; and although it's supported well on mobile devices, that support begins to falter on desktop browsers. Internet Explorer, in fact, has only supported it since version 10.

As such, although this seems like the most promising future-friendly method, it may not be suitable for your current website's audience. As with any other adaptive design method, first understand who you are building for.

Limber Up

At this point you have done all you can to responsively decide *what* JavaScript you are going to use; but *how* can you now use that JavaScript in a lightweight way that enables you to build mobile-first? If you have determined that you have JavaScript available on the client, you can begin to push things that would normally block page load to afterward. This can happen most easily via AJAX (asynchronous JavaScript and XML), and presents interesting design challenges when you're no longer moving between pages, but rather changing content within a page.

AJAX

If you have a heavy page that takes a long time to load, the simplest option is to just make a less heavy page. As pithy and obvious as that sounds, there's actually some unique and challenging work behind the idea.

When you are building mobile-first, you are defining the *core* experience, and after that what is just *nice to have* content. Why, then, would you bother loading anything more than that core at page load?

Conditional Loading

One of the largest challenges when building for the entire spectrum of devices is determining how to deliver the *same* content to all those devices, and their

connection speeds. While ensuring that the most important content of a page is reachable instantly, you can actually push the *nice to have* content later, and inject it via JavaScript.

What you use to do that is AJAX, asynchronous JavaScript and XML. AJAX is not one language, or a specific technology, but rather the combination of JavaScript with an `XMLHttpRequest` object to provide a data exchange between the browser and the server without a page reload. Despite the name, it does not actually *require* that the content being accessed is XML. You can pull back JSON data, or even full HTML and CSS.

These techniques help a *ton* when optimizing a page, but you need to be careful not to use them in such a way that you introduce more barriers to universal accessibility of a website through its URL regardless of device.

A great method to do this is an “AJAX-including” pattern that relies on JavaScript to enhance the way links to different content work. To demonstrate how it works, suppose you have the following markup:

```
<div id="content-block">
  <main class="main-content">
    ...
  </main>
  <aside class="complementary-content">
    ...
  </aside>
</div>
```

By default, and in the most heavy way, you would pull all the content for both `.main-content` and `.complementary-content`, but if you have determined that `.complementary-content` isn’t the core of what you’re trying to convey, but rather just nice to have, instead push it out of the page.

This takes a bit of work on the server side. First, you need a URL to just that content, let’s say “`/aside`.” That link would have a full HTML page with your website’s header, footer, and whatever other container you would like to display. Once you have that wired up, your first step can be rewriting your main page’s HTML, as shown here:

```
<div id="content-block">
  <main class="main-content">
```

(continues)

(continued)

```
...
</main>
<aside class="complementary-content">
  <a href="/aside">Complementary Content</a>
</aside>
</div>
```

However, now you have potentially created a worse experience, especially for users who have the bandwidth to pull everything in at once. To remedy that, create an additional version of your content on the server, a page that simply holds what would have been *within* that original `.complementary-content` container. You still need an end point for that, so call it `/aside-fragment`.

Using HTML5 `data-` attributes, you can reference that as follows:

```
<div id="content-block">
  <main class="main-content">
    ...
  </main>
  <aside class="complementary-content">
    <a href="/aside" data-content="/aside-fragment">Complementary
Content</a>
  </aside>
</div>
```

Your JavaScript work is to then reach this element, load the content from the `data-content` attribute, and replace the link with that content. The code to do this is as follows:

```
var $aside = $(''.complementary-content');
var $contentLink = $aside.find('[data-content]');
$aside.load($contentLink.attr('data-content'));
```

The `<aside>`'s link is removed, and replaced with the actual content, after the load succeeds.

A Note on Performance

Note one caveat, however. Every time you do this, you are incurring an additional HTTP request. Each AJAX-include can defer the time it takes for the core experience to be enhanced with this additional content. On mobile devices, this becomes

a serious problem, as the number of concurrent HTTP requests allowed by devices can be frighteningly low.

This limitation means that while the technique is still useful, pushing your *entire* site after page load just means you've redefined page load.

JavaScript Linking

Instead of pushing your *content* after page load, what if you could change the way users interact with links after they decide to learn more. Suppose you have the following markup:

```
<div id="container">
  <header>...</header>
  <main class="main-content">
    <p>Some content</p>
    <a href="/more">Learn More</a>
  </main>
  <footer>...</footer>
</div>
```

That Learn More link simply leads to more content on a new page. While browsers are getting better about caching what they can, assume that on that next page you repeated the header and footer—that seems wasteful. The following adds a `data-content` attribute again, but this time links to *only* the content that would replace the `.main-content`:

```
<div id="container">
  <header>...</header>
  <main class="main-content">
    <p>Some content</p>
    <a href="/more" data-content="/more-fragment">Learn More</a>
  </main>
  <footer>...</footer>
</div>
```

This way, instead of automatically loading in this element on page load, you can capture the link click, prevent its default behavior, and just load in the content. Check it out here:

```
$('.a[data-content]').on('click', function(event) {
  event.preventDefault();
```

(continues)

(continued)

```
var $main = $('#main-content');  
$main.load($(this).attr('data-content'));  
history.pushState(null, null, $(this).attr('href'))  
});
```

That second line specifies that the link should *not* follow its `href` attribute, as it would by default. Then, you're pulling in the `data-content` from the link and replacing your main content.

In that last line, you're updating the browser's URL to reflect the *actual* link to the content. Because you are preventing the link from being followed, you need to *explicitly* tell the browser where the content would be if users wanted to get to a page that shows the same thing just pulled in with AJAX.

Think of a normal link click, two major things happen on the client:

1. The page content changes.
2. The URL changes.

If you don't have either of those happening normally, you need to force them both manually so that if a user was to land on the *new* URL, they would expect and receive the *new* content.

Perceived Performance

Now that you've built up your toolset, consider for a moment your actual goal. Is it truly to make sites as quick and performant as possible, or is it to make them *appear* that way?

The perceived performance of a website is just what it sounds like: how quickly the site appears to perform a task. In many ways, humans are dumb and distracted by shiny, moving things, and you can use this lack of attention to your advantage by designing in such a way that, regardless of whether you can improve *actual* performance any more, you can use techniques to increase perceived performance.

An effective technique revolves around decreasing the time between a user's input and some form of visual feedback, such as a progress indicator. In fact, sometimes

adding the slight delay of a progress indicator can exploit the user's inability to accurately judge real performance, thereby making a website seem faster.

Therefore, let's finally put a few of the concepts from this chapter together, keep *perceived* performance as a goal, and see how you can design a highly interactive JavaScript-enabled web application.

The Train Tracker

You're taking on a new project: a subway tracking website. The website is obviously most useful on mobile devices, and should be as lightweight as possible. You have access to a city's routes and timing data, and you want the web application to do the following: Create a list of routes, each with its given collection of stops. For each stop, you query the city's timing data, and display a real-time ETA of when the train will arrive at that stop.

Distilling that project brief, you have three different views:

- Display the routes.
- Display the stops on a route.
- Display the real-time ETA at a stop on a route.

You could certainly cram all these views into a big list—you have all the data—but you want to work as lightweighted as possible and make your home page simply a list of color-coded routes as buttons (see Figure 5-8).

Although you know that you'll need any of the eight different routes, you delay loading any of them until the user explicitly chooses one:

```
<a href="/blue" data-stops="blue.json" alt="Blue Line">
  <i class="icon-train"></i>
</a>
```

Assume that you have populated JSON lists of all the route stops on your own server. If no JavaScript is enabled, the user is taken to a basic list of stops along that route, but because you successfully determined that JavaScript is available on this device, you can dynamically poll a simplified JSON end point to retrieve only the list of stops and then populate it in your page.



FIGURE 5-8

A subway-tracking website. All you populate at page load is the main list of route links.

An implementation of those links is shown in the preceding code snippet. An example of the JSON is shown here:

```
{
  "stops": [
    {
      "LINE": "blue",
      "STATION": 40890,
      "STATION_NAME": "O'Hare",
      "LON": -87.90422307,
      "LAT": 41.97766526
    },
    ...
    {
      "LINE": "blue",
      "STATION": 40390,
      "STATION_NAME": "Forest Park",
      "LON": -87.817318,
      "LAT": 41.874257
    }
  ]
}
```



```
}  
]  
}
```

With JavaScript, you ignore the link to `/blue`, and instead populate a portion of your DOM with the blue line's stops. However, that isn't an instant process.

Visual Feedback While Waiting

Add some visual feedback while the stops are being loaded: Change the background color and insert a simple spinning wheel progress indicator (see Figure 5-9).



FIGURE 5-9

Visual feedback of progress after a user selects the blue line but before its stops have been loaded

After the data has arrived, you hide the progress indicator and feed that data into the screen; the result is shown in Figure 5-10.

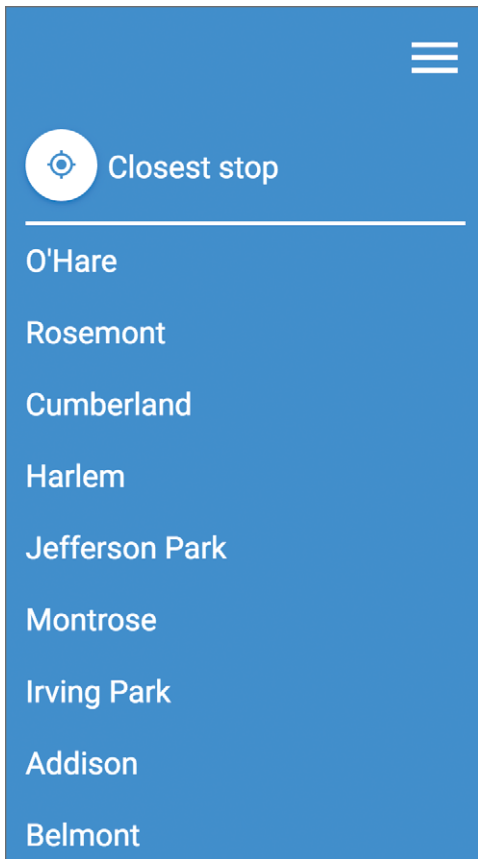


FIGURE 5-10

The line's stops have been loaded dynamically into the page without a refresh.

This is now a list of stops pulled in from the end point on your server, parsed, and spit back out onto the page as a list of links to their station IDs on the city's server that you have to poll. The next bit of markup shows a prototypical stop, followed by an example stop:

```
<a class="stop-link" href="/<%= LINE %>/<%= STATION %>" data-stop="
<%= LINE %>/<%= STATION %>"><%= STATION_NAME %></a>
<a class="stop-link" href="/blue/40890" data-
stop="blue/40890">O'Hare</a>
```

You can assume *some* server-side implementation of a “/blue/40890” end point is available for non-JavaScript situations, but let's again work toward the perceived performance of reaching another domain and pulling back data.

In this situation, not only is the content being conditionally loaded, but there's no reason to retrieve stop arrival data for an entire list of stops to begin with. The goal is sensible defaults, and a user is going to be looking for one stop, not all of them.

To handle a user clicking on a stop, again use JavaScript to ignore the link's `href` attribute, and this time reach beyond your server to retrieve ETA data. The chances are good that this will take longer than just getting JSON for the route's stops from your own server; so rather than simply add a progress indicator, put this content into a small new modal that's easy to exit or attempt to refresh the browser to feel some control when things begin to take too long. This is shown in Figure 5-11.

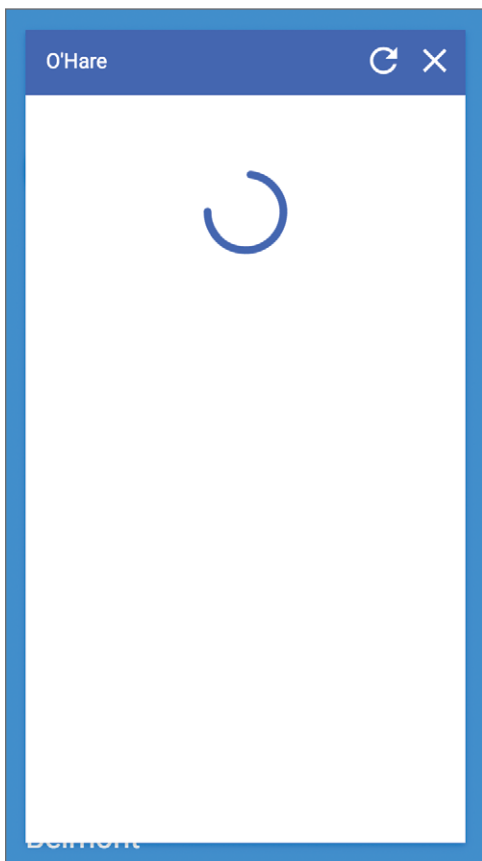


FIGURE 5-11

Progress indicator while content is being fetched and AJAX-loaded in for an individual stop's ETA information

Once the content is retrieved, you again remove the progress indicator and fill the list with parsed JSON. However, keep the close and refresh links for this stop at the top of the modal. Given that this is time-sensitive information, a user may want to refresh just this part of the page. Behind the scenes, this would do exactly the same thing that clicking the stop's link had just done. The populated modal is shown in Figure 5-12.

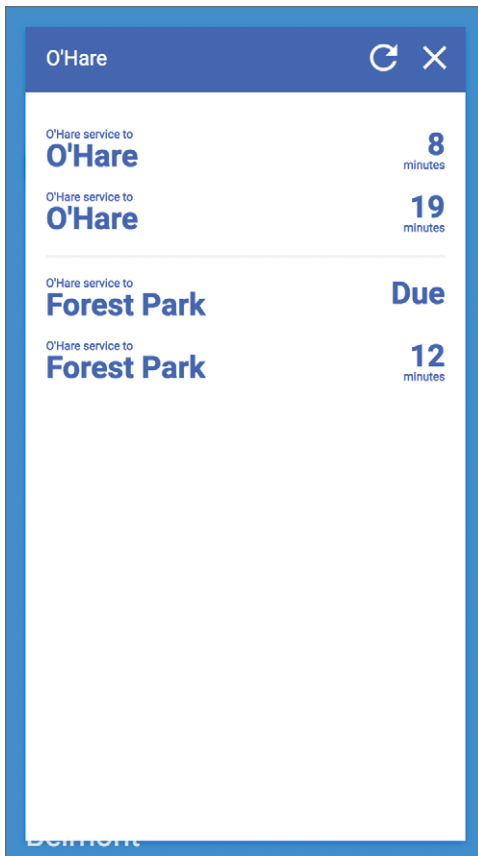


FIGURE 5-12

An individual train stop's data after successful loading

Just-In-Time Feature Detection

Now back up to the list of stops again for a moment. Note at the top of Figure 5-10 you have that additional link to closest stop. Plenty of devices, phones and computers alike, have access to the HTML5 geolocation API through the use of (surprise) JavaScript on the browser. Following is some simple code to use this API:

```
function getLocation() {  
    if (navigator.geolocation) {  
        navigator.geolocation.getCurrentPosition(alertPosition);  
    } else {  
        alert('Geolocation is not supported or has been disallowed.');    }  
}  
  
function alertPosition(position) {  
    var location = 'Latitude: ' + position.coords.latitude +  
                  'Longitude: ' + position.coords.longitude;  
    alert(location);  
}  
  
getLocation();
```

Open your developer console and try that code. Provided the API is actually available to you, you should first get a dialog asking if you are willing to share your location. Because geocoordinates *could* pose a safety risk, you have to handle not only the API not being available, but also users deciding not to let you use it.

This is a kind of just-in-time feature detection. You need to show the button by default, because you don't want to immediately ask for all kinds of permissions on a user's device as soon as they reach the website—especially if they're not even planning to use the feature.

Check out the JSON for your stops again. You have each stop's latitude and longitude coordinates. Using some math that you can easily find online (and that I don't need to add here to show off), you can identify the closest stop to a user by first attempting to get their location, and then run through the list to find the shortest distance.



NOTE

geolocation is one of many HTML5-specific APIs available in the browser. Examples of this and many others can be found at <https://github.com/AurelioDeRosa/HTML5-API-demos>.

However, only attempt to match a user's location to a stop if they actually click that link (see Figure 5-13).

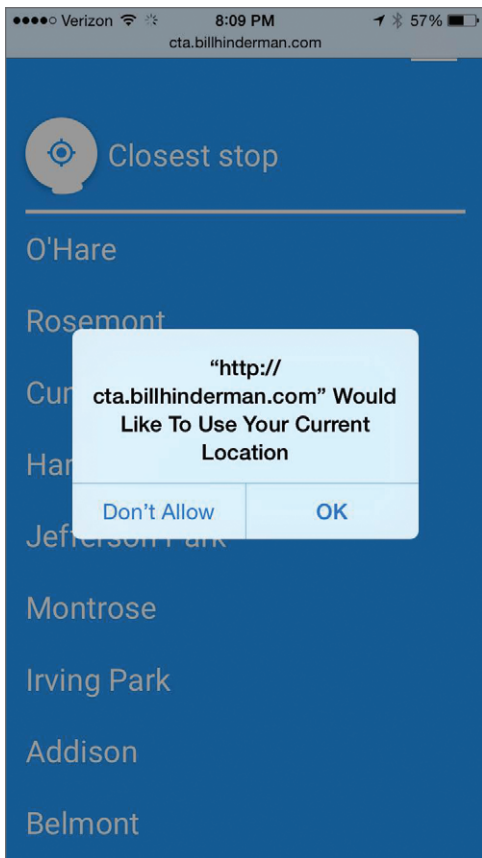


FIGURE 5-13

Just-in-time feature detection. The site asks for location services only if the user tries to use them. Notice the progress indicator on the Closest Stop button.

What if the user selects Don't Allow or is on a device without geolocation services? Because you opted to show the link by default, you need to indicate that the location services failed. However, consider perceived performance: To the user, there is no difference between waiting a long time and having a nonworking feature.

Therefore, add a timeout for geolocation of a comfortably long 15 seconds. After this amount of time, remove the option and display some kind of nice message that assumes the blame for the service not working.

Conversely, if it works, select the closest stop and do exactly what would be done if a user had clicked that stop to begin with: The modal pops up with the progress indicator, data is retrieved from your external data source, and then it is parsed and shown on-screen (see Figure 5-14).

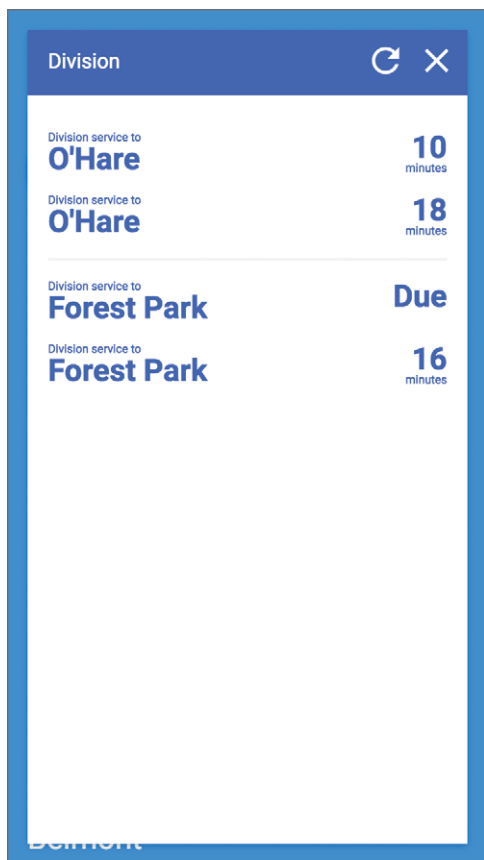


FIGURE 5-14

Successful location services used, and data loaded

And you're done. In the most simple scenario, you have a list of links that accesses pages with another list of links, which then accesses pages with lists of times. When you add JavaScript, you have a rich web application that dynamically pulls data into a page while giving users visual feedback that makes it seem lightweight and performant.

Check out the live version of this web application at cta.billhinderman.com.

Summary

You now have all the tools you need to make rich, mobile-first responsive websites. A sturdy foundation has been laid to begin exploring responsive and interactive data visualization built upon the following techniques:

- Progressive enhancement
- Sensible defaults
- Mobile-first CSS
- Media queries
- Responsive JavaScript
- Conditional loading
- Perceived performance

In the following chapters, you'll use all these tools to build visualizations that not only convey complex information, but also distill it to the most lightweight and simple presentation.

Try It

Want to practice some responsive JavaScript techniques?

- **Easy:** Check out the code on responsivedatavisualization.com/chapter-five/q1. You have all the assets ready to pull three different JavaScript files into your page, but are not yet pulling them in. Build the logic switch to pull `js_small.js`, `js_medium.js`, and `js_large.js` in at the correct sizes. A sample solution is also given in the repository.

- **Intermediate:** Expand on the preceding question by handling resize and orientation events within your logic switch. A sample solution is also given in the repository.
- **Hard:** Check out the code on responsivedatavisualization.com/chapter-five/q2. It's a page with one main section, and two optional sections. You'll notice other files containing JSON representations of the optional sections. Rewrite the `index.html` page to load only the main section by default, and push the loading of the other sections after initial page load. A sample solution is also given in the repository.
- **Expert:** Expand on the preceding question by only pulling the additional sections in when users request them. Explore different ways to make the performance of this operation *feel* faster to a user. A sample solution is provided in the repository.

TWO

PART

Creating Responsive Data Visualization

CHAPTER 06: [Data Design: An Abridged History](#)

CHAPTER 07: [Responsive Data Visualization Tenets](#)

CHAPTER 08: [Thinking Small](#)

CHAPTER 09: [Asset Dependence](#)

CHAPTER 10: [Code](#)

CHAPTER 11: [Building the Future-Friendly Web](#)

06

06

Data Design: An Abridged History

Until this point, you have not been considering how your content would affect your design. In this section, layout will still be a major contributing factor to how you design, but you will also begin to consider how your actual content (your data) impacts how you produce a product for end users. In this chapter, you'll take a brief look at a set of useful terms for thinking about data visualization on the responsive web.

It's important to understand the process of data collection and analysis to establish a baseline of what work needs to be done to find useful data, and what work needs to be done to turn big data *into* useful data visualization.

Learning From Data

Put your finger down on this paragraph and pause for a moment to look around you. There are concrete facts everywhere you look. You've touched one paragraph today. The dishwasher has been running for eight minutes. You have six pairs of clean underwear in the drawer.

These concrete facts are data. They lack assumptions or conclusions; they are simply things that you can observe and measure. Visualizing data can help you build information and draw knowledge from facts that may, on the surface, seem innocuous.

The things around you right now might not seem like they *need* visualization. They're fairly straightforward. But put them in context. Your dishwasher running for eight minutes *means* that there are still 12 minutes left in its rinse cycle, and then 10 in its dry cycle. You have guests arriving in 15 minutes, which *means* you'll need to spend seven minutes distracting them from the lack of wine glasses available. Adding more data creates more room for interpretation and finding *meaning*.

An even more visualization-friendly example: You're a gardener. You need to keep track of planting times, soil and water needs, pH levels, fertilization times, etc. You could certainly keep this data in your head, or spill it into a notebook as one long list. But a chart with each plant as the row and each of those points of data listed above as a column seems like an obvious solution to get to the *meaning* of: "What do I need to do in the garden today?"

We visualize data because, at a certain point, we lose track of the abstract things to which we're trying to attach it. Whether in a table, chart, or infographic, visualization of data helps move from "this is the data" to "this is what the data means." Distilling data into understandable representations enables us to glean greater knowledge than we ever could by just trying to remember all the information listed out. This process is called *data science*, and it can be broken down into the following steps:

1. Live in the world.
2. Observe things about that world.

3. Collect those observations as raw data.
4. Process that data.
5. Clean your data set.
6. Make exploratory analyses.
7. Build models around analyses.
8. Communicate/visualize/report.
9. Make deeper analyses from reports.
10. Make decisions.

This may seem like an overwhelming simplification of how to receive and handle data, which it is, but it's a handy list to look back to if at any point you find yourself getting bogged down by technical details.

Data, Information, and Knowledge

Data is collected from the environment in which you live. It is strictly raw findings and it is perfectly concrete. Through processing, data is distilled into information, which can be shared in a meaningful way. Finally, this information can be analyzed, and used to produce some kind of output or findings, which we'll call knowledge.

Looking back at the gardening example, two pieces of data would be:

The tomatoes were watered three days ago.

The tomatoes need water every three days.

Processing that data, the resulting information is: The tomatoes need water in zero days. Using your context, that you are the gardener, you can gain the following knowledge: You need to water the tomatoes today.

It's worth digging into data, information, and knowledge a bit more, as they provide the basis for how collecting, parsing, and conveying information to a user adds value to your data.

Data

So data is our least abstract concept. It's the stuff that we see and collect. The sky is blue. My arm is tingling. I taste oranges. My heart beat 120 times in the past 45 seconds. Figures 6-1 and 6-2 show images of two common signs that are nearly universally understood, with the first one designating a restroom for males and the second designating a restroom for females. In other words, they serve as the data representing useful (or in this case, necessary!) information.



FIGURE 6-1

Sign for a washroom your author should use



FIGURE 6-2

Sign for a washroom your author shouldn't use

The data in this scenario is as follows:

- There are two washrooms.
- The washrooms are somehow different.
- The person viewing the signs must select one.

Information

Signs such as these, and many others you see and respond to unconsciously every day, are effective because they convey information.

Figure 6-1, of course, is found outside of a men's room. The fact that there are urinals, a worse smell, and offensive graffiti are data. The sign itself is conveying information.

In this simple example, the data reflects the existence of two bathrooms. The *information* being conveyed is that one is a men's room, and one is a women's room.

Knowledge

As someone who has certainly used a public restroom before, you can apply your prior *knowledge* to select the correct door after viewing the signs, readily fulfilling whatever need prompted you to seek the restroom in the first place (e.g., wash hands, empty bladder, apply lipstick, etc.). Knowledge is what we get out of information when we put it in context. It is how we learn from the massive amount of data around us and apply those learnings to make our lives better.

Size Matters

The most simultaneously important and meaningless term in our field today is *big data*. It's important because it reflects the ever-growing volume of information that people want to manage, and it's meaningless in the sense that what is big today might seem small tomorrow. Chapter 11, Building the Future-Friendly Web, elaborates on that point.

Big data is, as simple as it sounds, a lot of data. It's an exciting term in our field because it takes a greater amount of processing power and design forethought to distill big data into unambiguous information that can be processed into meaningful knowledge. That being said, it is not *necessarily* more meaningful than small data.

It's safe to say that the amount of data available is going to keep growing as bandwidth gets cheaper, data collection becomes more universal, and the amount of time spent on web-connected devices essentially becomes 24/7. That being said, the value comes from gaining *knowledge* from data, not just having it.

That means that you will need to do even more work to process data into an expression that is both **effective** and **efficient**. As content creators for the web, we need to recognize that the amount of input we have is going to reach simply insurmountable levels for humans to manually parse.

Data's Data

Data itself can have its own data. That is, there can be concrete facts about a set of concrete facts. This is called *metadata*.

A pre-web example of metadata would be the card catalog of a library. It was standardized, which provided patrons with information about each of the library's resources in a standardized manner. A library's card catalog held a register of the materials (books, magazines, newspapers, graphics, maps, etc.) held within the library. The catalog would obviously not hold the materials themselves, since... that's what the library was doing, but would rather be the pertinent information for the library's patrons to seek out the actual material. That information would be: the author, the title, the subject, and the date of publication for a material. With this metadata, a patron could find the material they were looking for based upon the organization of the library itself.

The purpose of metadata is to provide for the discovery of relevant or related information. It comes in two different flavors: *structural* and *descriptive*.

- **Structural Metadata:** Structural metadata (or control metadata) is self-explanatory: It is used to describe the *structure* of some data—that is, the design

and specification of how the data itself is organized. An example of structural metadata would be a book's table of contents.

- **Descriptive Metadata:** Descriptive metadata, which is often more useful to us on the web, is used for discovery of related data. It is used to group items by, for example, author, subject, or keyword. On a website, descriptive metadata can often be found in the tags on a blog post or news story.

The Big Pile

Assume you have a big pile of data, and that data has a big pile of metadata. Walking through the data science process outlined at the beginning of the chapter, you first have to remove data that is not helpful. This can be referred to as cleaning the data, and really just means beginning to parse out the subset that is going to help you extract meaningful information. Once it's parsed, you need to begin to categorize exactly what the data is, in order to glean deeper meaning from the raw observations.

Your big pile of data is all of the planting and soil information from the garden at the beginning of the chapter. If your goal was to create a working schedule for what needs to be done in the garden, a piece of data that could be removed from the set during cleaning would be "You did not plant any green peppers this year." It may be useful in other contexts, but not this one. Remember that the goal of data science is to extract and convey meaning from a set too complicated to parse mentally.

Extracting Meaning

As you try to extract *information* from your *data*, it helps to classify what sort of values you are looking at.

Data visualization expert Stephen Few has described eight different types of *quantitative* messages that users can extract from a set of data, and the associated

graphical representations that most effectively communicate those messages. The following list, created by Stephen Few in his 2004 article, “Eenie, Meenie, Minie, Moe: Selecting the Right Graph for Your Message” (Perceptual Edge, 2004), explores each of these.

- **Time Series:** In a time series, the values displayed stand for a single variable, demonstrating how something has changed over time. An example would be a company’s rate of employee retention over a given period of time.
- **Ranking:** A ranking orders values in either ascending or descending order during a single period. An example might be the number of clients each account executive accidentally beat at golf over the past year.
- **Part-to-Whole:** Values in a part-to-whole relationship represent proportional divisions of a greater whole—for example, the amount of bourbon each member of the creative team drank out of the bottle hidden in the creative director’s desk.
- **Deviation:** In a deviation, values correspond to differences between two sets of values. An example might be the difference between the agreed-upon budget and the actual budget.
- **Distribution:** A frequency distribution’s relationship counts the values *per some interval* from lowest to highest—for example, the years in which a company’s growth was between 0–5 percent, 5–10 percent, and so on.
- **Correlation:** A correlation is the comparison between two different variables to determine whether they move in the same, opposite, or unrelated directions—for example, the number of hours an employee spends working versus that employee’s weight.
- **Geographic:** In a geographic relationship, a value is compared across a map or geospatial layout—for example, number of bald employees born in each state.
- **Nominal Comparison:** In a nominal comparison, values are simply charted by some related aspect, in no particular order—for example, commercial sales by television channel.

Communicating Meaning

Graphics *reveal* the meaning behind data. Data visualization, and the architecture behind it, should keep the following goals in mind:

- **Identify:** Find a set of data that is meaningful enough that conveying it to people will help them, and complicated enough that its meaning is not intuitive without some level of visualization.
- **Manipulate:** Parse and combine the data in such a way that the meaning behind it starts to become more clear.
- **Format:** Whether or not it needed manipulation, change the way in which you are accessing your data so that it is standardized for consumption.
- **Present:** Use your standardized formatted data to present some representation that helps convey its underlying meaning.

The idea is to approach the data in such a way that understanding its *meaning* becomes obvious and natural for a consumer. Of course, that's easier said than done.

When you're communicating data, it's important to consider its graphical display. Whether the data is a sentence explicitly communicating information or an interactive third model, you want to maximize both *efficiency* and *effectiveness*.

Efficiency of display means reducing complexity, noise, or unnecessary data in a visualization. An effective display should provide data in a way that eliminates any questions and avoids unnecessary ornamentation that could lead to incorrect assumptions or correlations between the information it represents.

Effectiveness of display means providing information in a way that is understandable, clear, and actionable. Effective displays should empower a consumer to make decisions and change behavior.

Graphical Displays

In his book *The Visual Display of Quantitative Information* (Graphics Press, 2001), Edward Tufte defines *graphical displays* as follows: "Excellence in statistical graphics

consists of complex ideas communicated with clarity, precision and efficiency.” He goes on to list nine principles for effective graphical display:

- Show the data.
- Induce the viewer to think about the substance rather than about the methodology, graphic design, the technology of production, or something else.
- Avoid distorting what the data has to say.
- Present many numbers in a small space.
- Make large data sets coherent.
- Encourage the eye to compare different pieces of data.
- Reveal the data at several levels of detail.
- Serve a reasonable clear purpose.
- Integrate closely with the data set’s statistical and verbal descriptions of a data set.

Base Forms

Data can be communicated in a number of ways familiar to all of us. Although these common, well-known forms may not seem particularly complex, even just the number representation or anecdotal sentence surrounding its meaning are forms in which data is communicated. Looking back to the gardening example, a number representation of data would be: “Days until tomatoes need water: 0.” A sentence conveying this information would be “Tomatoes need water today.”

The base forms of data communication are numbers, sentences, tables, graphs and charts, and infographics.

Types of Graphs

Let’s first dig into graphs (and slightly into infographics). The following sections briefly outline the major players that I’ve recognized as more than a little bit useful, and more useful than a lot of others.

Visualization can become complex, and cute, and “designy”; but it still has to communicate data effectively—and that’s what these graphs are good at.

Tables

A table is often how raw observed data is first distilled. It is a simple, if unsexy, way to begin to draw conclusions about different sets of data along a series of dimensions. Table 6-1 shows an example. As you can see in the example, items along the top axis are compared along dimensions labeled on the left axis.

TABLE 6-1

A simple table

	Set 1	Set 2	Set 3
Dimension 1	5	2	6
Dimension 2	9	7	4
Dimension 3	1	0	5

Bar Charts

A bar chart (see Figure 6-3) presents grouped discrete data with rectangular bars. The lengths of the bars correlate to the values of the data they represent. Bars can be plotted vertically or horizontally.

Stacked Bar Charts

A stacked bar chart (see Figure 6-4) is similar to a standard bar chart, except each bar is divided into subparts to show both a part-to-whole representation of values and the differences between the wholes themselves across the sets.

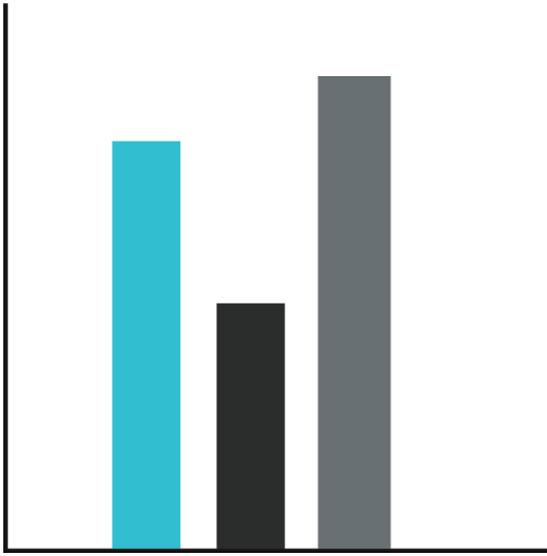


FIGURE 6-3

A bar chart. Sets are described along the x-axis, while their measure is communicated on the y-axis.

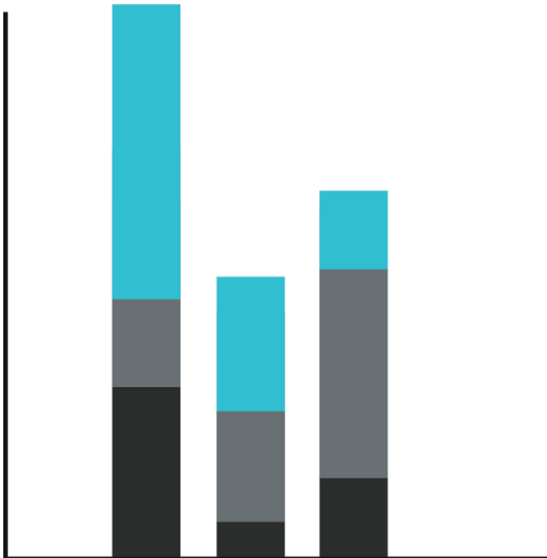


FIGURE 6-4

A stacked bar chart. Both the sets themselves and the parts within them are represented.

Histograms

Using a histogram (see Figure 6-5), you can represent the distribution of continuous data. Data in a histogram must be put into consecutive, non-overlapping “bins” in order to construct the series of bars in which each value is represented. The bars are touching to indicate that the value is continuous. A data set for which a histogram might be useful is the distribution of exam grades for a class.

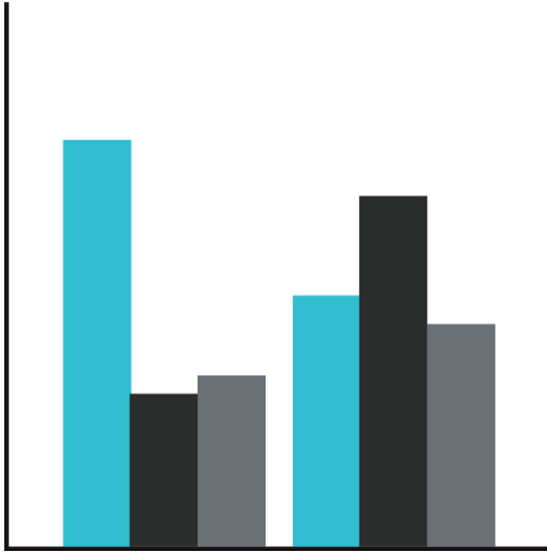


FIGURE 6-5

A histogram. Like a bar chart, but for continuous data on one set

Pie Charts and Donut Charts

A pie chart (see Figure 6-6) shows the relationships of multiple values as part of a greater whole.

A donut chart (see Figure 6-7) functions in exactly the same way as a pie chart, but it has a big blank spot in the middle (making it look like a donut). Ancillary data can be shown in this center spot, but it is typically used as is, rather than the pie chart, simply because the relative ratios of each section are more intensely obvious than in a pie chart.



FIGURE 6-6

A pie chart. Each slice makes up a part of the whole.



FIGURE 6-7

A donut chart

Scatter Plots

A scatter plot (see Figure 6-8) displays values for two variables in a data set. While it can be used similarly to a line graph (without the line) for a relationship whereby one variable is independent and one dependent, the greater strength of a scatter plot is the capability to expose nonlinear relationships between variables.

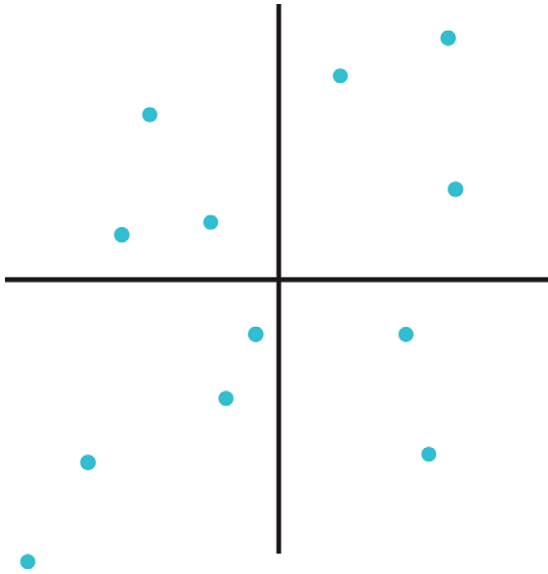


FIGURE 6-8

A scatter plot. Two variables can be plotted against each other to identify otherwise unknown patterns and relationships.

Moreover, if a mixture of simple relationships represents the data, these relationships become visually evident as patterns arise in the plot.

Line Charts

A line chart (see Figure 6-9) shows the relationship between a quantitative variable and a time series. It is essentially a scatter plot with a more obvious pattern, shown by straight line segments joining each point.

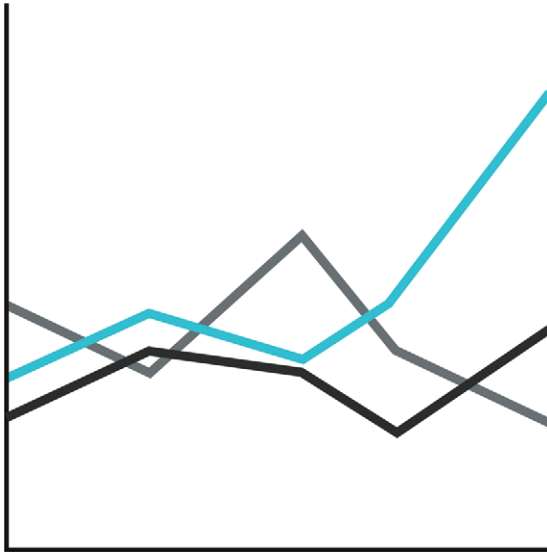


FIGURE 6-9

A line chart. The y-axis represents a measurable variable, while the x-axis represents a time series.

Area Charts

An area chart is similar to a line chart in that it shows the relationship of one variable to time. However, it differs in that its measuring method is typically more akin to that of a stacked bar graph, with the area from the graph's start to the measured point providing the meaningful information. The cumulative area *under* the line is the major conveying factor of an area chart. The area chart comes in three flavors: basic area, stacked area, and 100% area. The basic area chart starts all lines from the x-axis, like any other graph, and overlays them. The stacked area places areas of greater value on the bottom layer and “stacks” those with smaller values on top. The 100% area chart is similar, but data is stacked on top of each other and the data values for each item will total 100%. That is, the chart will always take up all of its available rectangular space.

A stacked area chart, which is likely what you have seen before, is shown in Figure 6-10.

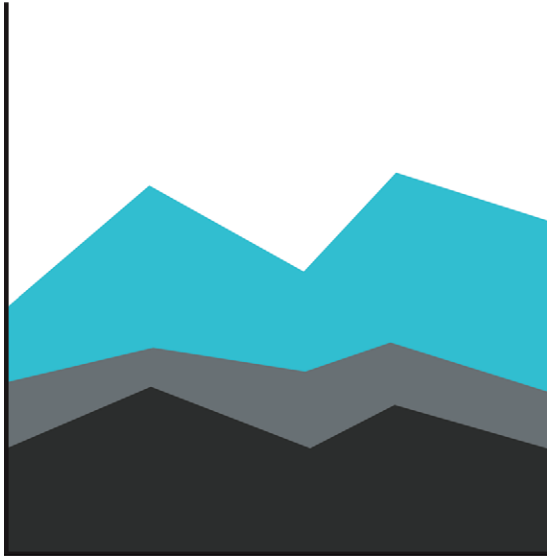


FIGURE 6-10

A stacked area chart. Cumulated totals over time are shown.

Dorling Maps (Bubble Charts)

A Dorling Map, or bubble chart, if you want to sound less fancy, is a variation on the scatter plot. The bubble chart represents variables in multiple ways: the x,y coordinates of a bubble, the bubbles' relative locations to one another, and each bubble's size.

Additional information beyond these dimensions can be effectively shown in a bubble chart as well, either by shading of colors within one set of data, or coloring different sets of data in uniform fashion and comparing their relative distances or sizes from one another as in Figure 6-11. The value gained by a bubble chart is the amount of different dimensions that can be shown in a relatively simple visualization.

Although called a map, it has nothing to do with geographical information. The Dorling map below could represent the overlap in mutual friends between two users on Facebook. The colors represent who the individuals were friends with first, the size being the amount of overlapping friends *those three* have, and the distance from either axis being the frequency of posting on one another's wall.

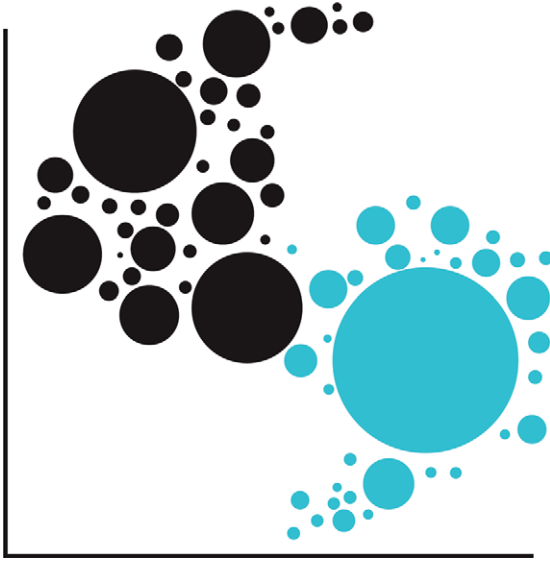


FIGURE 6-11

A Dorling map. It can show at least three dimensions of data, based upon size, clustering, and the x,y location of a bubble.

Venn Diagrams

A Venn diagram, or set diagram (see Figure 6-12), shows the relationships between two or more sets. Each circle represents a set, and items within the circles belong to the sets. The overlapping section in the middle contains items that belong to *both* sets.

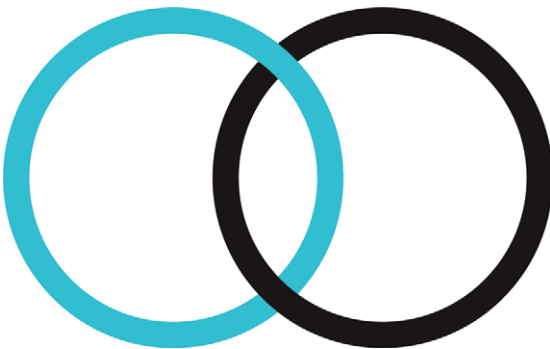


FIGURE 6-12

A Venn diagram. Two sets of data and their overlapping items are represented.

Nested Venn Diagrams

Similar to the Venn diagram, the nested Venn diagram (see Figure 6-13) shows the items belonging to increasingly more exclusive sets. If the largest circle is the *least* exclusive, each smaller nested circle represents a meaningful *subset* of its parent.



FIGURE 6-13

A nested Venn diagram. The largest circle is the least exclusive set, while each smaller circle represents a subset.

Networks

A network diagram, or node-link diagram (see Figure 6-14), represents the vertices and edges of a graph, showing the relationships between the vertices.

Networks can be either undirected (as in the figure) or directed, which means that whatever variable is being measured is valid only in a particular direction.

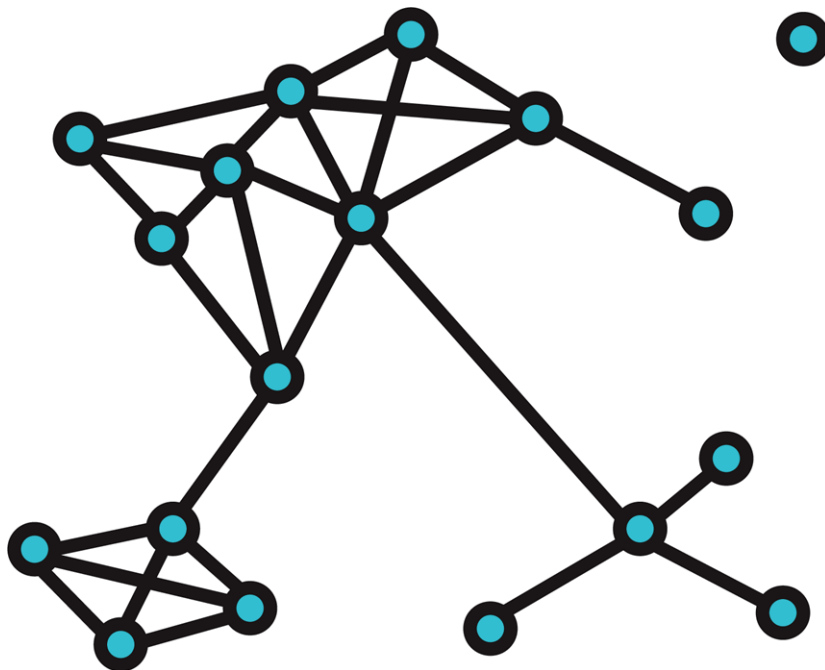


FIGURE 6-14

A network diagram. Relationships between vertices (the dots) are shown as edges (the lines).

Cartograms (Maps)

A cartogram (see Figure 6-15) is a map. It maps data to geographical locations. A standard geographical map is a cartogram; it's simply mapping the geographical data of a location onto its geographical location. More interesting cartograms substitute another category of information instead. This could be crime rates per county, electoral votes in a state, or traffic patterns on a highway.

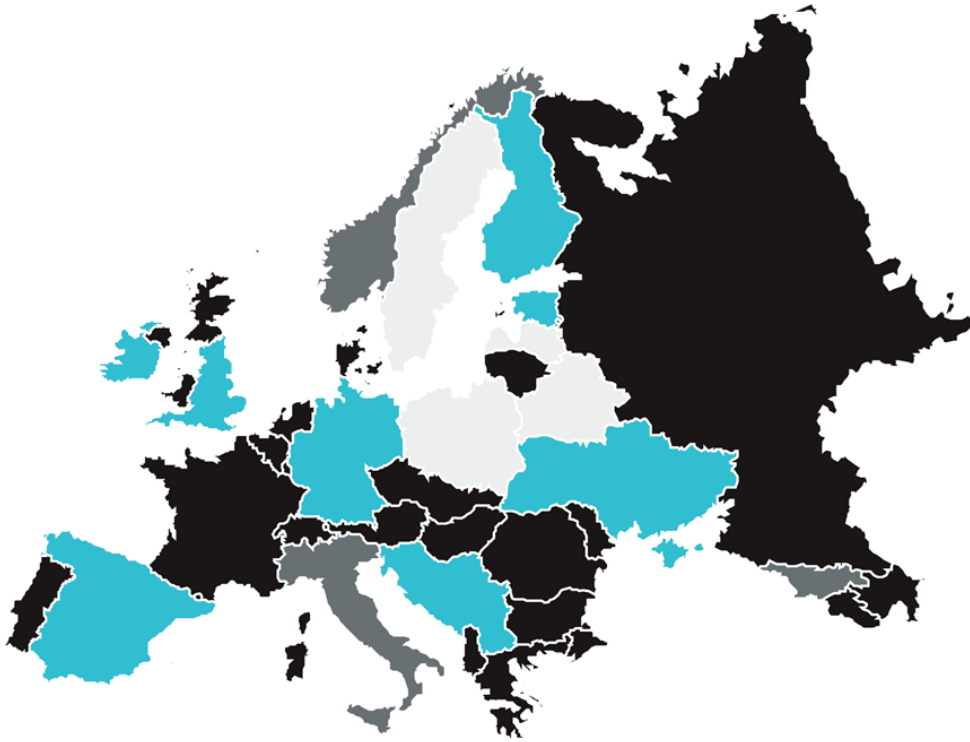


FIGURE 6-15

A cartogram. It's a map, but shows a different kind of information.

Radar Charts

A radar chart (see Figure 6-16) shows multivariate data. Multivariate data is data with multiple unrelated variables being compared to one another. The radar chart superimposes each of these variables onto the same chart to give greater insight into patterns found in places where they overlap. All axes must start from the same point (typically 0) at the center and extend outward. The position and angle of the axes typically don't convey any additional information other than the numbers themselves.

Radar charts are useful in both identifying trends among multivariate data, and also putting a spotlight on obvious outliers in the data, visualized as dramatic spikes or valleys on one axis.

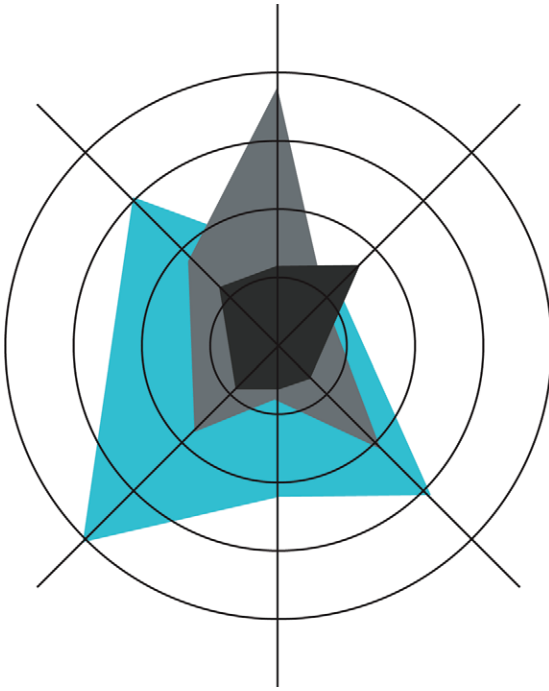


FIGURE 6-16

A radar chart. Multiple variables can be charted from the same initial point to determine where patterns may exist among multiple data sets, and whether there are any obvious outliers.

Tree Maps

A tree map (see Figure 6-17) displays hierarchical relationships between values. Each “branch” of the tree is represented by a rectangle, with smaller rectangles subtiled within other rectangles.

In addition to showing possible correlations between dimensions, size and color can be used in conjunction with the base construction, creating a much more space-efficient visualization technique, and showing larger data sets much more legibly than other types of graphical displays.

Heat Maps

A heat map (see Figure 6-18) shows the clustering of repeated data. Rows, columns, cluster size, and color can all be used to categorize different sets of values.

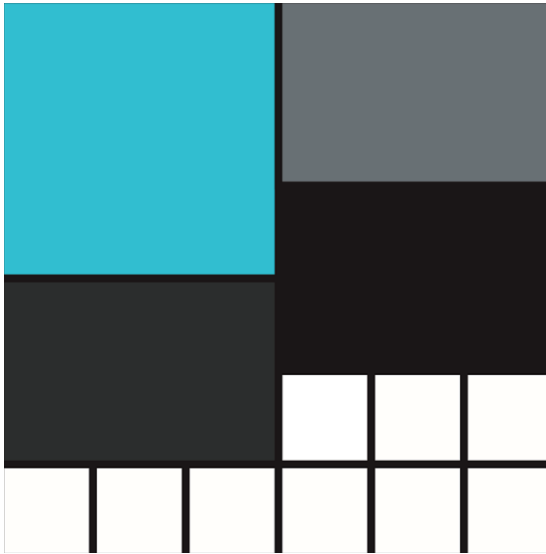


FIGURE 6-17

A tree map. Data that is more prevalent takes up more space.

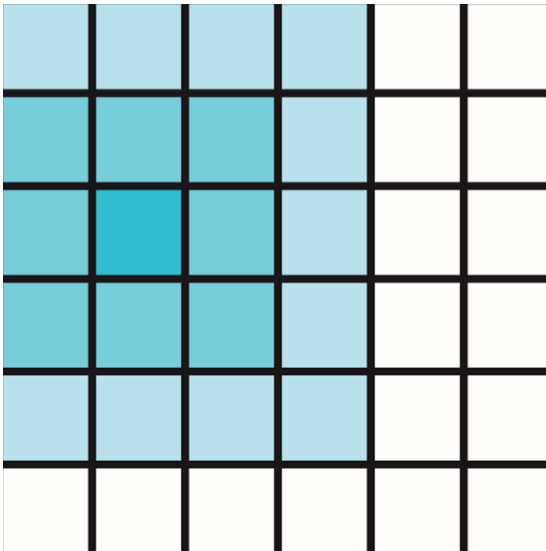


FIGURE 6-18

A heat map. Data points that cluster are shown with greater “heat.”

Timelines

In a timeline (see Figure 6-19), events are shown as points along a straight line from time A to time B. They can additionally vary in size or shape to signify importance, number of people involved, or some other secondary measure, but the primary function is to show when events happened chronologically.



FIGURE 6-19

A timeline. Events are shown in chronological order.

Sentences

“It is going to rain tomorrow.”

This last one is important. If you want to share an *obvious and critical* piece of information from your data, then there is no need to complicate the matter with a graph of some kind. Simpler is *always* better.

Of course, you have plenty of other ways to visualize data; but as you get into more esoteric visualizations, they begin to pull from one or more of the aforementioned types, and begin to work against the fourth principle of effective graphic displays: Induce the viewer to focus on the substance, rather than the methodology, the graphic design, the technology of production, or something else.

What You Get from the Web

So far, nothing put forth in this chapter is specific to the web, or even to digital data visualization. Each of the preceding graphs can be used to communicate data on a sheet of paper. Web design was born from, and informed by, print design; and,

just the same, web data visualization was born from print data visualization. However, there are web-specific aspects of the field that are both helpful and headache inducing.

Interactivity

Print visualization is a one-way street. Things are created, and things are consumed. There's no way to provide feedback on a sheet of paper, unless . . . you're making a data visualization flipbook, I suppose. (Give me credit if you happen to pursue that horrible idea.)

On the web, you do not need to stop at the first level of presentation. You have links, hover states, a keyboard, touchscreens, geolocation, and a wealth of other ways in which you can interact with the individual consuming your visualization.

Choosing a Sensible Default

In perhaps the most obvious way, you don't need to limit yourself to a single method of visualization. Sure, in print you can create two different graphs and put them near each other, but online you can present your information in unlimited ways at once.

For example, suppose you have a set of data that provides interesting insight when shown as a scatter plot, but also when visualized on a timeline, and each individual mark on the timeline also serves as a communicative pie chart. Recall how you handled the base cases for your responsive design.

You can pick a sensible default to display to your users, say, the timeline, because why not? Then, provided you have the page real estate, and bandwidth, show options to move to the other visualizations as well.

Meta Information

If you were working with print, you are limited to presenting the data, throwing any metadata into an appendix. Online, however, that's not the case. For example,

just as a blog post's tags can be used to relate it to other blog posts, you can attach metadata to your visualizations to create meaningful relationships between entirely different sets of data.

Remember from earlier in the chapter, the purpose of metadata is to provide for the discovery of relevant or related information. Structural metadata provides non-visual data consumers with information about how your data should be consumed, while descriptive metadata describes what the data is describing.

A Taste of Responsive

As you've now learned the basics of data communication through visualization, it's time to begin connecting this concept back to responsive design. If the goal is to always have an experience that seems natural on the device available, your methods for visual communication are not going to ever simply be "pick the right chart once." In Chapter 8, you'll learn about building an efficient API (application programming interface) to your data. That is, a way to reach your data without any screen.

But before getting there, think about what you're trying to solve for now: You need to account for a sea of screen sizes and device features, but you also need to account for any level of data complexity.

Providing Natural Experiences

The whole pile of different data visualization types described in this chapter were included for a reason (apart from how pretty they look). With so many possible ways to present users with information, it can be daunting to determine which is the best for your particular scenario. The bottom line is to provide the most natural experiences for users on any device.

That means you need to be efficient and effective with your presentation; and you can achieve that by adhering to sensible defaults and working responsively, displaying options and meta information only when it actually makes sense to do so.

Reliably Everywhere

The URI is the core of the web. In Chapter 2 you read about content parity between devices when reaching the same website. The “Thematic Consistency of Resource Identified by a URI” sums it up nicely:

This is a realization of the One Web principle, whereby content should be accessible on a range of devices irrespective of differences in presentation capabilities and access mechanism. Web sites may paginate their content in various ways corresponding to differences in device characteristics; therefore the navigation structure of the site, and possibly its technical realization, may vary according to the device class that is being served.

W3C MOBILE WEB BEST PRACTICES 1.0

Users want to do everything everywhere. Give them what they’re looking for, and don’t penalize or pardon them for the way in which they’re reaching your content.

This is just as important in terms of data visualization. It is understood that you cannot show the *exact* same dashboard on a mobile screen as you can on a high definition television. What matters is that you can still make the underlying *data* the same, and reachable.

A New Dimension to Respond To

Responsive data visualization has a dimension that you didn’t need to worry about in web design: the size of the data set. Apart from screen size, bandwidth, device class, or anything else, you need to first and foremost serve your actual data.

We’ll get into the weeds in subsequent chapters, but the point here is to understand that you need to be able to recognize and respond to data sets whether they contain seven entries or seven million.

In the next chapter, you’ll approach the task just as you did in Part One of this book, and begin to think of data visualization mobile-first.

Summary

What you've learned in this chapter is how to begin consuming data, parsing it into some useful information, and conveying it in an efficient and effective way to communicate meaning. At the start of the chapter, you learned the basic structure of *data science*:

1. Live in the world.
2. Observe things about that world.
3. Collect those observations as raw data.
4. Process that data.
5. Clean your data set.
6. Make exploratory analyses.
7. Build models around analyses.
8. Communicate/visualize/report.
9. Make deeper analyses from reports.
10. Make decisions.

Understanding how to handle the massive amount of data available to you is a pivotal skill in creating visualizations that are meaningful and communicative.

In the following chapter, you are going to revisit many of the tenets of responsive design that were introduced in Chapter 2, applying them to data visualization. You'll also learn a few new ones that were brought up at the end of this chapter, surrounding a new concept: responding to data.

0

7

07

Responsive Data Visualization Tenets

As you push toward building more complex, data-driven applications on the web, it's important to understand the many parallels between responsive data visualization and responsive design as a whole. In this chapter, we'll revisit some of the core tenets established throughout the book so far, adding clarity and specificity to how they relate to data visualization. You'll also learn some new principles for guiding how you build visualizations for the web.

Designing Content-First

It wasn't too long ago that creating a chart on the web was essentially impossible. In Not So Distant History, the horribly resistant path of least resistance was to use the .NET Framework to render a chart on the server, and return a static image via an interpreter layer.

You read that right: a static image.

The only way to have *interactive* charts in Not So Distant History was to force users to leave the browser and open Excel. Users were stuck looking at a JPEG of a chart, and it was virtually impossible to draw any meaning out of the data other than by actually *looking* at it without interacting with it, because the visualizations were simply images—there was no semantic markup embedded within them.

While images were a simple way of visualizing data, developers also turned to Java Applets and Flash. These were plugins that could render beautiful complex information loaded from external sources. Just as with web design as a whole, Flash seemed to dominate the market for a long time, as it removed the limitations of working strictly with the DOM.

However, that freedom ultimately caused their failure—they were so removed from any semantic concepts, and so heavy to implement, that better standards began to crop up. Flash is now mainly used to serve streaming media, ads, and interactive multimedia content on the web, and even those are moving toward native video implementations.

Thankfully, things have gotten better. Microsoft introduced Vector Markup Language (VML), originally for their Office Suite, as an XML-based file format to represent two-dimensional vector graphics. Although this standard didn't last, it did give birth (more accurately, was a fraternal twin) to Scalable Vector Graphics (SVG).

SVG is also an XML-based vector graphic format. It supports interactivity and animation, and is itself supported by nearly all major browsers. And *because* it is XML-based, writing an SVG is just like writing any other markup on the web. To that end, SVG provides an incredibly lightweight way to visualize information. Not only that, but it uses tools you are already familiar with, and does not depend upon anything outside of what is built into the browser already.

Still Just the Web

Here's the good news: You're not starting from scratch. Not even a little bit. This book is about building responsive data visualization *for the web*, so you're still working *with the web*.

What that means in a practical sense is that the tenets put forth in Chapter 2, and everything you've learned since then, still apply. Responsive data visualization within the browser can be thought of as a continuation of the more broad responsive design. You'll see a lot of the terminology from Chapter 2 repeated in the following sections, but taking on slightly more specific meanings in how they apply to data visualization.

Just be sure to keep it top-of-mind that you are not working with any new visualization-specific tools, and are speaking the same language you have up until this point.

It's All Greek to Me

Throughout this chapter, you'll notice the element `<viz></viz>` in some of the code. This is not a real HTML element, but simply in place to encapsulate your visualization within the page's layout. It serves to show that you are still working with elements on a page; and just as you would style a `<div>` or a ``, you can apply those same principles to a `<viz>`.

We'll get there, I promise. For the time being, though, think of the data visualization as a small object model within your DOM. It will have HTML elements that can be styled with CSS and modified with JavaScript. All the implementation details will make sense once you lay a base of higher-level techniques.

Everything's a Nail

That said, the techniques described here will seem very familiar. There's a lot of "Hey, remember when..." in this chapter because responsive data visualization is a *subset* of responsive design, not another topic.

You're going to use the same four principles defined earlier, but now you'll connect them directly to how you build web applications backed by large sets of data, not just pictures of kittens and angsty blog posts.

Distilling Data

What is the key difference in your input to a data visualization, as opposed to a text-driven website? Well...the data. Rather than content living on the page, you're consuming, parsing, and displaying complicated sets of numbers in order to convey that set's implicit meaning clearly and efficiently.

Distilling your data into something presentable to the user is built upon the same foundation of progressive enhancement that fuels responsive web design. You'll first retrieve raw data. Then, you'll find some way to standardize that data and parse out a meaning. That standardized data becomes the content you build into meaningful, semantic markup. On top of that markup, you then apply styling that makes the visualization more clear.

Back to step one, though: Where do you get your data?

Handling Data

Data can come from anywhere. For the purposes of this chapter, we'll focus on two typical sources:

- Your own server
- An external API

The difference between the two should be apparent from their names, but let's be thorough. The first data source enables you to change the structure of your parsed data because you have total control over it. The second is out of your control, pulled from a database over which you have no control.

Both of these situations will be common as you work with data visualization on the web, so it's important to learn techniques that work for either case.

Structured Data as Content

For the bulk of this book, you have been considering the mobile user as your most limited and basic case. You consider a small screen with a bad Internet connection and limited functionality, and build for that; then you enhance. We've been calling this progressive enhancement.

It's been fine to limit your approach in this way when talking about responsive design—after all, you're designing for humans. But if you're thinking about communicating data sets to these same users, then you need to first consider a user with no screen at all. You need to consider the browser itself as your base user. Instead of working mobile-first, you need to work API-first.

An API, or application programming interface, is a set of protocols specifying how software components should interact. Or, in human terms, it defines how two pieces of software should speak to one another. An API, in your case, should be the expected inputs and standardized outputs of interacting with your data set.

Recall for a moment the subway schedule app from Chapter 5 (see Figure 7-1). In your project brief, you had three distinct views:

- Display the routes.

- Display the stops on a route.

- Display the arrival times at a stop on a route.

Remember that you decided to push any of your data loading to after page paint to improve the app's perceived performance. After a user clicked a route from the initial route list, you loaded a blank page with a progress indicator while the data was being fetched. The link to the blue line was constructed as shown in the following code:

```
<a href="/blue" data-stops="blue.json" alt="Blue Line">  
  <i class="icon-train"></i>  
</a>
```

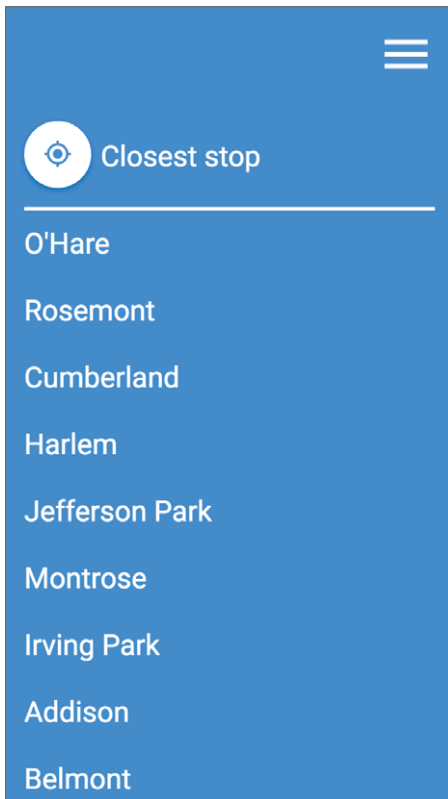


FIGURE 7-1

The train tracker website you constructed in Chapter 5

During this time a call was made to your API. Assume you have the simplified code for that call in the following code. This code is part of a larger JavaScript application that uses Backbone to construct views; but for your current purposes, you just need to know that the first code block populates a collection of stops, while the second code block is the model for an individual stop.

```
define([
  'underscore',
  'backbone',
  '../models/stop.js'
], function(_, Backbone, StopModel){
  var StopsCollection = Backbone.Collection.extend({
    model: StopModel,
    url: function() {
```



```

        return this.apiUrl;
    },
    initialize: function(models, options) {
        this.apiUrl = options.apiUrl;
    },
    parse: function(data) {
        return data.stops;
    },
    });
    return StopsCollection;
});

define([
    'underscore',
    'backbone',
], function(_, Backbone) {

    var StopModel = Backbone.Model.extend({
        defaults : {
            'DISPLAY_URL': '',
            'LINE': '',
            'STATION': 0,
            'STATION_NAME': '',
            'LAT': '',
            'LON': '',
        },

        initialize: function() {
            this.set({'DISPLAY_URL': '#/' + (this.get('LINE')) + '/' +
                (this.get('STATION'))});
        },
    });
    return StopModel;
});

```

You'll notice that you reference `apiUrl` in the stops collection. In this case, you were simply reaching a URL on your server that had a list of all train stops on that route. Therefore, your API input was simply the correct route's name. For example, the blue line's would have looked like this:

```
var apiUrl = './lines/blue.json';
```

Remember from Chapter 5 that your API's response was JSON formatted, as shown here:

```
{
  "stops": [
    {
      "LINE": "blue",
      "STATION": 40890,
      "STATION_NAME": "O'Hare",
      "LON": -87.90422307,
      "LAT": 41.97766526
    },
    ...
    {
      "LINE": "blue",
      "STATION": 40390,
      "STATION_NAME": "Forest Park",
      "LON": -87.817318,
      "LAT": 41.874257
    }
  ]
}
```

That works very well, but your app's URI to get to the blue line's stops (for humans) is `cta.billhinderman.com/blue`. There's a disconnect between your data URI and your human URI. Instead, construct your `apiUrl` as follows:

```
var apiUrl = './blue.json';
```

Because you are simply hitting your own server, you have the power to change this without too much effort: All it requires on your server is moving where you keep your stops lists to the top-level folder.

Now, apart from the type of file you're looking for, you have achieved continuity between your browser's computer-facing URI and your human-facing URI.

You'll learn about communication with other apps later in the chapter.

Build Basic Markup

So what have you just constructed? Your URI to `/blue` doesn't exist yet, but `/blue.json` is now all of the data you have parsed and decided is a meaningful representation of the stops along the blue line.

You have built your API for stops, and can move onto the actual markup. This becomes an exercise in building mobile-first, as you would do normally. The details of this design were a major focus of Chapter 5, but let's revisit the markup you created. Remember that your JSON representation has the following attributes, which you have decided are useful:

- Line
- Station
- Station name
- Longitude
- Latitude

Previously, you ignored the latitude and longitude in the example, as you were just walking through an exercise demonstrating how to populate a list with AJAX. Now, assume that you included it in your JSON for a reason, and produce the markup that is most meaningful to your users. Your more robust code is shown in the following example:

```
<li itemscope="" itemtype="http://schema.org/SubwayStation">
  <a id="stop-<%= STATION %>" href="#/<%= LINE %>/<%= STATION %>"
  itemprop="name"><%= STATION_NAME %></a>
  <span itemprop="geo" itemscope="" itemtype="http://schema.org/
  GeoCoordinates">
    <meta itemprop="latitude" content="<%= LAT %>">
    <meta itemprop="longitude" content="<%= LON %>">
  </span>
</li>
```

The link in the second line should look familiar. It was the basic markup that you needed to display which stop this was to a user, and to create the correct link to then get that stop's ETA information via the station ID. What you've added is called *meta information*.

This is the basic information that, rather than a human, and rather than the browser, another website can read. In this case, you're using the schema.org guidelines for structured data to let search engines understand that the element enclosed in the `` tag is a subway station.

The additional `itemprop` on the link sets the station's name to the content within that element; and all of the new information within the `span` is the geolocation of the stop, which you provide as latitude and longitude.

Schema.org structured data can be thought of as a type of semantic markup for elements that don't strictly have an analogue to print media. Whereas headings, paragraphs, footers, and asides are all supported tags in HTML, there obviously isn't a `<train station>`. When you add this additional metadata to the element, you are essentially describing it to a user that has no eyes—namely, a search engine crawler.

Throw Pretty at It

At this point, you've constructed the following:

- **API:** Your JSON file, which can be accessed easily by hitting its URI of `/blue.json`
- **Human-readable markup:** The actual tags that now show up on `/blue`, which link to each stop's ETA list
- **Website-readable markup:** The meta tags on your markup, which allow, among other things, search engine crawlers to parse out your information in a way more meaningful than a big list of text.

At the moment, however, your list of links looks something like what is shown in Figure 7-2.

You previously created some mobile-first, responsive styles for this website, and followed a good practice of keeping your markup (and classes and IDs) led by content, rather than the other way around. However, you did not call that out explicitly. You won't walk through that again, but instead use it to look at a related point.

As you build data visualizations on the web, you should be even more aware than in responsive designs of a blog or an online store, that the information you are sending across can become incredibly heavy. Later in the chapter you will learn about data size, this new property of responsive data visualization, but for now you can think about the next step in your train tracker app.



FIGURE 7-2

The train tracker without any styling

A user has clicked on an individual stop, and is waiting to get ETAs loaded into a list. You already know that you have to account for the user's device capabilities, but now you also have to account for (at least) the following cases:

- No trains
- No trains in only one direction
- Construction delays on a track
- Station closures

In addition, there will likely be times when something in your system actually goes wrong and you can't fetch data. All these scenarios lead to a concept called layering,

whereby data may be presented to the human user or a digital user (such as a search engine crawler), but it can also be incorporated into design techniques to modify how you present the data.

Let's look at that last item in the list: station closures. You likely wouldn't want to wait until a user tried to fetch ETAs for a stop to have that information. For now, assume some magic happens in the backend, and you make an external call to populate your JSON list with a new attribute: status. A stop would now be represented as the following JSON:

```
{
  "stops": [
    {
      "LINE": "blue",
      "STATION": 40890,
      "STATION_NAME": "O'Hare",
      "LON": -87.90422307,
      "LAT": 41.97766526,
      "STATUS": "CLOSED"
    },
    ...
    {
      "LINE": "blue",
      "STATION": 40390,
      "STATION_NAME": "Forest Park",
      "LON": -87.817318,
      "LAT": 41.874257,
      "STATUS": "NORMAL"
    }
  ]
}
```

You could then modify the link in your markup to take a class based upon this status, as shown here:

```
<li itemscope="" itemtype="http://schema.org/SubwayStation">
  <a id="stop-<%= STATION %>" class="status-<%= STATUS %>" href="#/<%=
LINE %>/<%= STATION %>" itemprop="name"><%= STATION_NAME %></a>
  ...
</li>
```

Then, in your CSS, add the following rule to display to the end user that this station is currently not in operation:

```
.status-closed:after {  
  content: " : Not operating";  
}
```

You may be thinking that this isn't actually visualization in the sense that it doesn't look like a chart.

As explained in the previous chapter, the point of any data visualization is to *reveal* the meaning behind data. You should always strive for the goals of identifying, locating, manipulating, formatting, and presenting data in such a way that it communicates its *meaning* easily and effectively.

You certainly could plot these stations on a map, and then create a timeline showing how far each train is from a station on interaction (and, spoiler, you're eventually going to do that just for fun), but the simple text that you have created here is ample visualization to communicate the meaning of your data.

Revisiting Responsive Design Principles

In this section we'll start to connect this all together. At the beginning of this book, four guiding principles were defined for working responsively on the web: be universal, be flexible, be economical, and be specific. To adapt those to data visualization, these principles are now: Be universal, be flexible, be economical, and be specific.

That's right.

The big secret is that responsive data visualization is simply responsive design with more factors to *respond* to. Your goals, your guiding principles, remain the same. No matter what you're conveying to a user, you want it to be as accessible and natural an experience as possible. Knowing that, we can focus on how the four principles defined in Chapter 2 apply to data visualization.

Be Universal

When we first started looking at responsive design, we went back a decade to consider what it meant to be on the Internet. More than likely, users were at a desk, looking at a screen that was slightly bigger than their head. The typical user had a mouse and a keyboard, a relatively stable Internet connection, and no other Internet-connected devices.

Of course, by now we've established that this is no longer the case. Your mantra is to include as many *humans* as possible, but is that enough when the information you're presenting is no longer a blog post or an image?

Achieving Content Parity in Data Visualization

Now that you're dealing with data, you can't suddenly assume that users only want to see certain things on certain devices. Remember: If someone can get to your content, you should understand that to mean they want to get to it. It's more important to make your data available than it is to ensure that your design is pixel perfect.

However, a visualization *demand*s a certain layout, and you can't simply use the same display with some responsive shift to fit it nicely onto a screen.

More important, you can't move in the opposite direction and *hide* data simply because certain features or the space are not available. The problem is that text and data just don't work the same way.

Your solution actually sits at a layer below the visual design, or even the human-visible content. To achieve content parity in data visualization, you have to ensure that when users who don't even have a screen reach your website, they receive *all* relevant and meaningful information.

What Is the URI to Data?

I've referenced the "Thematic Consistency of Resource Identified by a URI" a few times by now, but it's worth bringing up again in this instance:

This is a realization of the One Web principle, whereby content should be accessible on a range of devices irrespective of differences in presentation capabilities and access mechanism. Web sites may paginate their content in various ways corresponding to differences in device characteristics; therefore the navigation structure of the site, and possibly its technical realization, may vary according to the device class that is being served.

W3C MOBILE WEB BEST PRACTICES 1.0

Before, this quote was fodder for the argument against a separate mobile web, and for responsive design. Basically, mobile users want to do everything that desktop users can, and the URI should always point to the same information regardless of a user's device type.

As mentioned in the “Distilling Data” section of this chapter, you can go a long way toward the concept of data content parity by building “API-first.” In the train tracker example, you laid out your hierarchy of data presentation as API, human-readable markup, and website-readable markup, but let's now explicitly define each step:

1. **Raw data:** This is your input—all the unsorted and yet-to-be meaningful data from which you are hoping to draw conclusions.
2. **Parsed data:** This is the data that you have scrubbed and begun to analyze.
3. **API:** This is the first layer that you need to expose—your most basic way of *showing* data to a user. This can be, for example, JSON output of a collected subset of your parsed data.
4. **Visible markup:** Fed by the data in your API layer, and achieving parity with that data, this is the HTML markup that constructs your visualization.
5. **Metadata markup:** This is the invisible layer of markup that helps provide additional context for the data presented in your visible markup.

Table 7-1 summarizes how exactly you achieve parity between the visible layers of data for both machine and human end users.

TABLE 7-1
Layers of Data in a Web Application

Layer	Data Set	Visible to User	URI
Raw data	All of your data	No	-
Parsed data	Organized subset of raw data	No	-
API	Meaningful subset of parsed data	Yes	/data.json
Visible markup	Meaningful subset of parsed data	Yes	/data.html
Metadata markup	Meaningful subset of parsed data	Yes	/data.html

The most important point of this table is that the three layers visible to a user all sit at the same URI end point, and convey the same information.

Be Flexible

When you worked toward this principle earlier, it meant learning to steer away from pixel-precise designs that work only at pixel-precise sizes.

At this point, it should be clear that layouts need to be flexible to accommodate the massive and ever-increasing variety of screen sizes and device capabilities for which you are building. However, when it comes to visualization, you have to first reevaluate what you can *actually* be flexible about it terms of layout. Similarly, you must also think about flexibility in your data set.

Forget the Device

Figure 7-3 shows a very small sample of the large variety of devices currently available. In terms of data visualization, although you can *optimize* for the device features that you anticipate will be used (or even know will be used) by your largest audience, you clearly can't count on any one screen as being the only size you have to consider (or, say, one phone, one tablet, and one laptop all made by Apple).

Thinking about the amount of screens on which your visualization can potentially be accessed, you must begin to consider the level of granularity, placement of labels, and overall beauty vs. clarity of your designs. Your data should always be available, but it doesn't always need to be immediately visible. Because visualizations invite

interaction much more than, say, a paragraph of text, you have the freedom to *layer* more data on top of your initial and most simplified view.



FIGURE 7-3

A small fraction of the screen sizes for which you build

Forget the Data Set

Here's the flip side, though. Your data isn't going to be perfectly tailored to the screen size you have, or the visualization technique you choose. Outside of hand-crafting every visualization for every update to every data set, there will be cases where things don't work as well as others.

This is your second glimpse at the new factor you need to respond to: data size. All your data is available at the API level, and it should be *reachable* at the markup level, but you can use techniques of grouping and layering (more on that soon) to present the clearest version of the data by default.

You'll hopefully have connected this to the progressive enhancement concept of *sensible defaults*. Whereas on a larger scale a sensible default can refer to the toggled

state of a menu, or the initial display of addendum content, you can now apply it to data visualization.

To do that, you need to massage your displayed data such that based upon its complexity, scale, and importance, the most obvious conclusions can be drawn immediately; more complicated information can be added upon interaction.

Consider the example of an online travel website selling flights. In this case, when a user searches for a round-trip flight, you want to return some useful predictive data regarding whether or not you anticipate fares for that route increasing or decreasing within the next week.

You have a massive data set of the route's past prices over the past two weeks, and projections as to what that route's price *might* do with varying levels of confidence. On top of that, your raw data has prices, airline carriers, number of seats remaining, and alternate routes.

What does the user want, though?

The user wants to book a flight, so this is all just ancillary information to them. It may help you push immediate conversion (if the fares are likely to go up), but otherwise it is not a user's main goal. Therefore, make your sensible default as displayed in Figure 7-4.

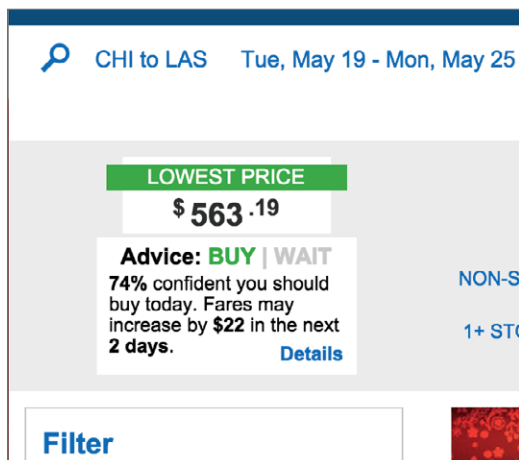


FIGURE 7-4

A summary view of flight prediction information

Here you have a small module on a page that provides a simple message indicating, as a percentage, how confident the user can be that the current fare is the best price available. That is the clearest way to display this information. You can also provide interaction with more detailed, robust data, which users could access by clicking the “details” link to launch a modal displaying the chart shown in Figure 7-5.

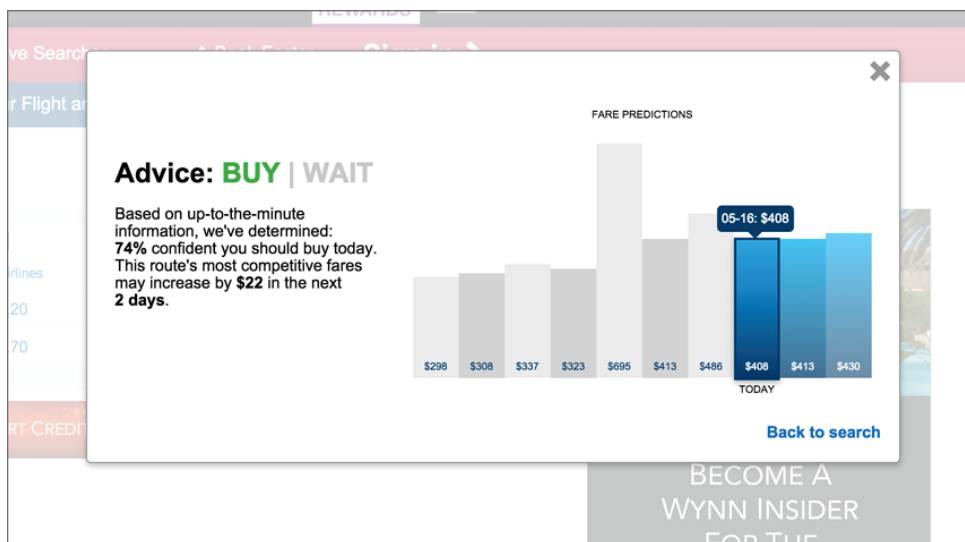


FIGURE 7-5

A detailed view of flight prediction information

In this case, users *deliberately* interact with the module because they want to see more information. You now can layer on the next level of detail.

Be Economical

The principle of being economical means that you should serve users the most efficient experience possible. In terms of data visualization, this principle doesn't translate into specific best practices but rather a general awareness of just how much you are serving to a user, as every bit of data increases the page's weight.

Performance Is Design

Statistics (from the GSMA) show that only 20% of N. America customers have LTE enabled handsets. Globally, the LTE penetration rate is ~5%. 3G networks are still the norm, not the exception.

BURSTING OUR BUBBLE: MOBILE PERFORMANCE OUTSIDE THE FIRST WORLD

You saw this statistic in Chapter 2, but it's just as relevant here. Creating efficient HTML, CSS, and JavaScript are one thing, but achieving performant web experiences when you are parsing and displaying complicated data requires waging a much more important battle.

Actual Performance

There are some things you can do to improve the *actual* performance of your data visualizations. First, and hopefully most obvious, is stick to standard HTML elements.

Earlier in the chapter I mentioned some of the old ways of displaying rich interactive visualizations on the web: Flash or other plugins. These can become incredibly heavy, however, not to mention their reliance on installing a separate piece of software to begin with. This is the easy part.

What's more of a challenge is scrubbing your data until what you're asking for is the bare minimum of what you need. Any piece of unnecessary information within your data set that you're still requesting will take time and resources to receive, parse, and ultimately throw away. Therefore, ensure that you have a reason for everything within your set.

The third way to improve actual performance is to front-load your design plans around only the information you'll actually need for initial page load, and after interaction. Defining this enables you to group that information into separate, lightweight API calls.

Perceived Performance

When you were walking through responsive JavaScript, you spent a lot of time in the train tracker example pushing information loading past page load using asynchronous JavaScript and XML (AJAX). In that case, you had a relatively simple set

of data to retrieve your list of stops, and were pushing your request to an external API to receive ETAs for an individual stop after load in order to give the user immediate feedback while the bulk of their request was handled in the background. In Figure 7-6 you see the loading state displayed to the user immediately.

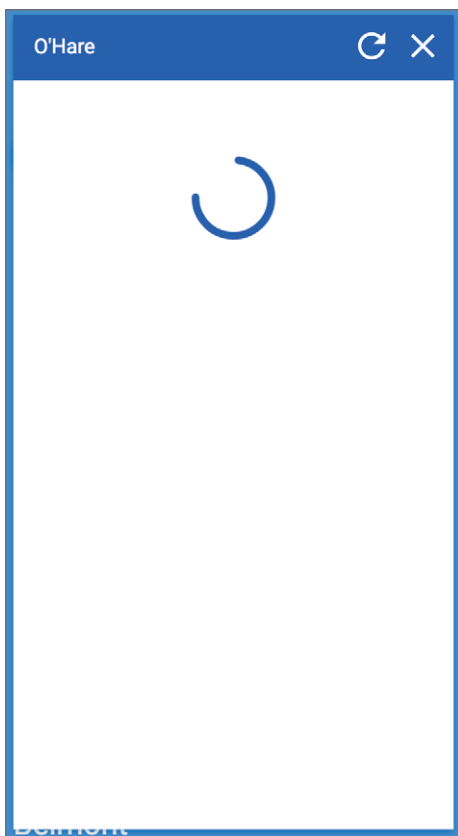


FIGURE 7-6

Progress indicator showing a loading state in the train tracker

As your data becomes more complex, and your parsing more robust, you will encounter situations in which an extended amount of time is required in order to pass data back to the user. This duration quickly becomes unacceptable if users are staring at a blank page. Perceived performance enables your program to always look like it's doing useful work, even if that work isn't visible.

AJAX Loading

Let's look back at the air travel example. Remember that your data set included (at least) the following:

- Route origin
- Route destination
- Departure date
- Return date
- Two weeks of pricing history
- One week of pricing projections
- Confidence level in each prediction
- Seating information in each prediction
- Overall buy/wait recommendation
- Overall pricing prediction
- Confidence level in overall prediction likelihood
- Alternate routes

If all you want to do is display buy/wait, likely price jump, likely price date, and overall pricing prediction confidence, then there's a *massive* amount of data you don't need. Now, you're hitting an external API, so you can't scrub the data or change the payload you receive as output, but you can limit what you parse out for this initial call.

As it's not page-critical information, you can let the page load, and *then* make your call to the API. When you receive the entire payload back, you can first look at the overall buy/wait recommendation. If it's wait, you can stop right there and not worry about building any markup.

If it's buy, you'll then get the following:

- Price jump

- Likely price jump date
- Pricing prediction confidence

Once you have this information, you can build your module and populate it on the screen; and although you've saved the rest of the model you receive from the API call, you haven't yet bothered to do anything with it.

AJAX Interaction

That is, until the user clicks. Referring back to Figure 7-5, you're showing the bulk of the information there. If this were a call on your own server, you could have separated it out entirely from the first call, and not bothered waiting to get the payload back; but for now, you've just kept it saved on the client.

Now that the user wants to interact with the information, parse your data for each individual day's price, and build that into a new object list called `days`:

```
{
  "days": [
    {
      "DATE": "04/15/2015",
      "PRICE": 401,
      "TYPE": "HISTORY"
    },
    ...
    {
      "DATE": "04/21/2015",
      "PRICE": 519,
      "TYPE": "PREDICTION"
    }
  ]
}
```

The preceding JSON shows a simplified version of what data could be stored in your `days` list. From here, you can write the following code to construct the graph. Assume inputs of the `days` list, and a reference to today's price within that list:

```
var $graph = S('<div id="air-graph"></div>');
for (var i = 0; i < days.length; i++) {
  var day = days[i];
  var ratio = (day.price / today.price) * 100;
  var $dayMarkup = $('<div class="graph-column graph-' + day.type + '"
data-size="' + ratio + '">\
```

(continues)

(continued)

```
                <div class="graph-popover">'+day.date+':  
$'+day.price+'</div>\n                <div class="graph-info">'+day.price+'</div>\n                </div>');  
    $dayMarkup.css('height', ratio+'px');  
    $dayMarkup.appendTo($graph);  
}
```

As you can see, you're doing a lot more here than simply getting the prediction. The beauty is that you didn't bother doing any of it until the user explicitly indicated what was wanted.

A quick reminder while talking about AJAX: Working on your design in the browser not only lets you easily spot where your design falls apart, it also forces you to see just how much of an impact changes to your data set, API, parsing logic, and design have on your overall performance.

Be Specific

Finally, remember to be specific with the ways in which you optimize your data visualization. Remember Edward Tufte's nine principles for effective graphical display:

- Show the data.
- Induce the viewer to think about the substance rather than about the methodology, graphic design, the technology of production, or something else.
- Avoid distorting what the data have to say.
- Present many numbers in a small space.
- Make large data sets coherent.
- Encourage the eye to compare different pieces of data.
- Reveal the data at several levels of detail.
- Serve a reasonable clear purpose.
- Integrate closely with the data set's statistical and verbal descriptions of a data set.

Once you have built a visualization that is universal and flexible, optimize toward the preceding principles in order to create concise, beautiful charts. This differs

from the larger responsive design in that rather than just trying to present what you want to present, you are now hoping that *through* your design, users will be able to glean further knowledge and dig deeper into data in ways you had not considered.

The Web Is a Continuum

Your content is going to be everywhere. In order to ease the process of revealing meaning behind data in your visualizations, recognize opportunities where you can detect a user's browser features and offer a shortcut or an additional layer of interaction to maximize *efficiency* and *effectiveness*.

What that means in a practical sense is exactly what it means in terms of the larger responsive design: Don't build for devices; build for features. And don't *force* an experience, but *enhance* the existing one when possible.

Know Your Data

Although you shouldn't limit your users, and therefore won't ever know the wealth of devices they're using to consume your data, you (hopefully) know your data. You know if it's a big complicated set of things, or only a few facets being compared—or even if filtering can allow both of those situations.

Specificity in this sense means beginning to treat the characteristics of your data set as its *features*, just like those on your devices, and then recognizing and responding to those features.

Seven Points of Focus

Chapter 2 defines seven points of focus that became your guide for building responsive web design skills in the following chapters. Those points were: measure, flow, nesting, end points, breakpoints, typography, and images. In this section, you'll

recycle a few and add a few others: measure, flow, end points, breakpoints, images, grouping, and layering.

Measure

When you looked at responsive design tenets in Chapter 2, the first point of focus, measure, mentioned using relative units, rather than pixel values, to describe a layout. If you had a layout described in the markup and CSS as shown in the following example, your design would be static, and forever stay at that size, as the layout on the left of Figure 7-7 illustrates.

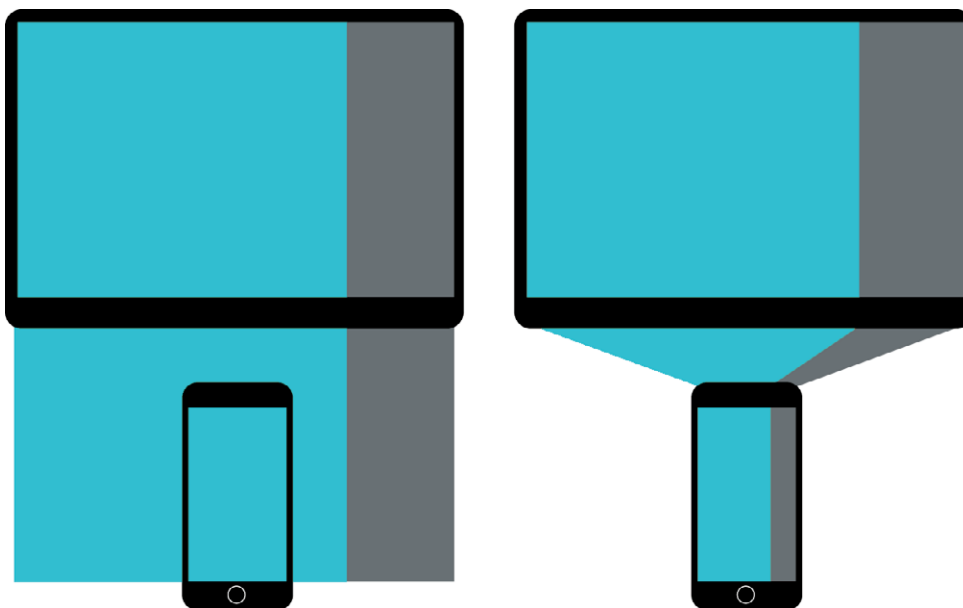


FIGURE 7-7

A layout with static (left) and fluid (right) measure

```
<body>
  <section class="section-content">
  </section>
  <aside class="section-sidebar">
  </aside>
</body>

.section-content,
```

```
.section-sidebar {
  float: left;
}
.section-content {
  width: 768px;
}
.section-sidebar {
  width: 256px;
}
```

However, if you change to the relative units highlighted in the following code, then your layout will always take up 100% of the screen's width, no matter what it is. This is reflected in the layout on the right of Figure 7-7.

```
.section-content,
.section-sidebar {
  float: left;
}

.section-content {
  width: 75%;
}
.section-sidebar {
  width: 25%;
}
```

You could apply the above CSS to the following markup and have the same layout as if the content wasn't a visualization:

```
<body>
  <viz class="section-content">
  </viz>
  <aside class="section-sidebar">
  </aside>
</body>
```

However, a more meaningful application of flow is using comparative and easy to modify ratios to describe your data. Remember from the flight price predictions graph that you held a reference to the current price, and constructed a ratio for each other day. That snippet is repeated here:

```
for (var i = 0; i < days.length; i++) {
  var day = days[i];
  var ratio = (day.price / today.price) * 100;
```

(continues)

(continued)

```
var $dayMarkup = $('<div class="graph-column graph-'+day.type+' "  
data-size="'+ratio+'">...');  
...  
$dayMarkup.css('height', ratio+'px');  
}
```

It's much harder to apply relative measure to visualizations backed by actual numbers. What this enables you to do is respond to device screen sizes with a modifier value that keeps the scale of your visualization the same.

Remember your very important formula in defining relative sizing?

$$\text{Target} \div \text{context} \times (\text{base size}) = \text{result}$$

In this case, target was your element, context was its parent element (or the screen), and result was the relative size. What you modify this to for your visualization calculations is as follows:

$$\text{Target} \div \text{comparator} = \text{ratio}$$

The target is the element that you are styling.

The comparator can be one of two things:

- The total, as in the case of, say, a donut graph
- The element that you select to represent a 100% base size

In the flight example, you used the second option. Your representation of the current day's price was what you wanted all other prices compared to, so you built your ratio off that price, normalized to 100px tall.

The base size is a representational value that you use to determine the size of the element displayed. It can simply be 100%, or, in the case of the flight example, it can be mapped to a pixel value. This is the only part of the equation that you will be able to change responsively and keep an accurate ratio.

Flow

In the greater responsive design, flow is the way in which content moves within your layout. The following code represents a static layout that doesn't reflow, as shown in Figure 7-8.



FIGURE 7-8

A static website that doesn't properly reflow

```
<body>
  <header class="section-header">
  </header>
  <viz class="section-content">
  </viz>
  <footer class="footer">
  </footer>
</body>

.section-header {
  height: 200px;
  position: absolute;
  top: 0;
  width: 100%;
}
.section-content {
  height: 600px;
  position: absolute;
  top: 200px;
  width: 100%;
}
```

(continues)

(continued)

```
.footer {  
  height: 100px;  
  position: absolute;  
  top: 800px;  
  width: 100%;  
}
```

Using the following CSS instead allows it to flow, as shown in Figure 7-9:

```
.section-header {  
  position: relative;  
  width: 100%;  
}  
.section-content {  
  position: relative;  
  width: 100%;  
}  
.footer {  
  position: relative;  
  width: 100%;  
}
```



FIGURE 7-9

A website with proper flow

The best part of considering this flow is that it enables you to let your content determine the shape of your layout. Rather than force a box around what your user is trying to access, the information itself drives the website's overall design.

When you're talking about responsive data visualization, there is no difference on the layout side. If you encapsulate your visualizations (for the time being) within a `<viz>` element, you can treat them like any other block element.

Where it differs is within the data. Charts don't reflow and shift as nicely as text does, and eventually there just isn't enough room for the data to fit on a screen. This is where layering, flowing in the z-axis, comes into play. Whereas previously you considered the width and height within CSS for layouts, you can now decide how elements stack on top of one another too. You let the content hierarchy within the visualization determine the importance level of the z-axis.

End points

End points in responsive designs are the maximum and minimum values to which you want your layouts to flow. The following code shows the use of a set of end points on your `.container` to prevent it from growing or shrinking indefinitely (see Figure 7-10). The corrected result is shown in Figure 7-11.

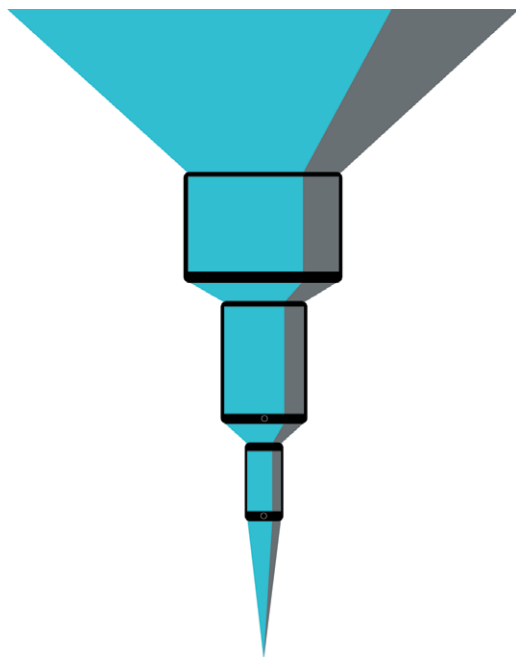


FIGURE 7-10

A fluid layout with no end points

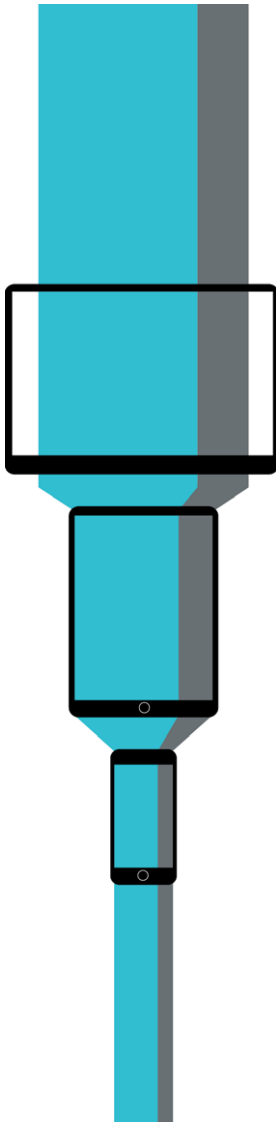


FIGURE 7-11

A fluid layout with end points

```
<body>
  <div class="container">
    <viz class="section-content">
  </viz>
```

```
<aside class="section-sidebar">
  </aside>
</div>
</body>

.container {
  margin: 0 auto;
  max-width: 960px;
  min-width: 320px;
}
.section-content,
.section-sidebar {
  float: left;
}
.section-content {
  width: 75%;
}
.section-sidebar {
  width: 25%;
}
```

As with flow, there is really nothing unique to data visualization here. There will be a point at both the upper and the lower end of screen sizes where your visualization breaks down.

The new definition of end points, though, is whatever the upper and lower limits are to the data that makes sense within a given type of chart. That is, at what point do you either not have enough data, have too much data, or have *grouped* enough of your data that it no longer makes sense to display in a certain way?

Breakpoints

How you solve for those end points, you'll recall, is by adding media queries within your CSS to create breakpoints at which you apply new rules. Figure 7-12 provides a quick refresher of what breakpoints look like. The code follows.

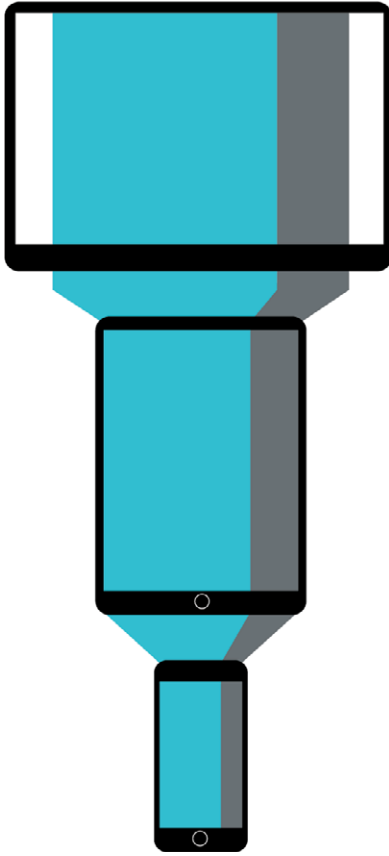


FIGURE 7-12

A breakpoint set at mobile changes this layout

```
<body>
  <div class="container">
    <viz class="section-content">
    </viz>
    <aside class="section-sidebar">
    </aside>
  </div>
</body>

.container {
  min-width: 320px;
}
@media screen and (min-width: 768px) {
  .container {
```

```

    margin: 0 auto;
    max-width: 960px;
    width: 95%;
  }
  .section-content,
  .section-sidebar {
    float: left;
  }
  .section-content {
    width: 75%;
  }
  .section-sidebar {
    width: 25%;
  }
}

```

However, what you learned in the preceding section brings up an interesting problem. With less complicated content, you can just keep adding breakpoints when layout goes a little haywire.

With data visualizations, however, there reaches a point where the entire charting technique you are using may not make sense. That can result from both screen size constraints and data size constraints. If you're drawing with code, then you can add classes and attributes to elements just as you would any other HTML element, and affect them with media queries; however, there's more to do when you're redrawing the chart. The way you solve for this is to add a listener for the viewport resize event:

```

$(document).ready(draw);
$(window).resize(draw);

function draw() {
  var w = $(window).width();
  var h = $(window).height();

  // Call re-draw functions
  if(w > 768) {
    unDraw();
    draw768();
  }
  ...
}

```

(continues)

(continued)

```
    else {  
        unDraw();  
        drawDefault();  
    }  
}
```

In the code, you listen for the page's size, then call an individual function that *undraws* the chart you currently have (if any), and then draw the one you want based on the screen size.

It's trivial, then, to come up with a technique to draw based upon data changes as well. Check out the code that does that and notice the similarity to your viewport listener:

```
function draw(data) {  
    // Call re-draw functions  
    if(data.items.length > 300) {  
        unDraw();  
        drawLargeSet();  
    }  
    ...  
    else {  
        unDraw();  
        drawDefault();  
    }  
}
```

You are now responding directly to how many items are within your set.

Grouping Points

I've mentioned this concept a few times throughout this chapter already. Grouping is a technique to get to more meaningful insights from data when it is either of the following:

- Too complicated
- Too small of a data set

It has a close analogue to the idea of a breakpoint, but whereas the breakpoint is conceptually the point at which the entire chart should be redrawn, the grouping point is one at which certain related items in the data should be shown as one item (or split back apart) in a default view.

Images

The bulk of what was outlined in Chapter 2 about images can be summarized as responsively pulling in different rasters when screen size and bandwidth allow, and switching to using vector images that scale indefinitely.

As you may have assumed, SVGs are the go-to option for responsive data visualization. They're lightweight, built with actual markup that can be semantically tagged, and scale however you want.

That being said, there are some other options.

Responsive Rasters

The first option is exactly what you walked through in Chapter 2's Images section. Instead of relying on vectors to scale for you, you can instead use breakpoints to send different sizes of an image to different screen sizes. This works best in your CSS when you are using background images:

```
#myViz {  
  background-image: url('viz1_sm.png');  
}  
@media screen and (min-width: 768px) {  
  #myImg {  
    background-image: url('viz1_lg.png');  
  }  
}
```

Remember the drawback, though. At 768px and greater, you are forcing the client to download both images. This solves the *visual* problem, but you take a performance hit.

To better solve this, you can use the RESS (server-side responsive) techniques outlined earlier in this book to send different sizes of an image to different classes of screen without sending the extra bloat. The one piece of obvious overhead to this technique is that you have to make and store all those different sizes of images.

Thankfully, you have scripts for that.

Plugins and Scripts

Ignoring the plugins you have already debunked (Flash, Shockwave, applets), there is a wealth of scripts that run on the server that can rapidly produce image representations of data on the client. These take the grunt work out of building differently sized images for each screen size.

The biggest failure of using raster images, though, is that they are inaccessible to anything other than a visual user. Remember from Chapter 2 that rasters (like a png or jpeg) are just rectangular grids of colored pixels. To the markup, to a screen reader, to a search engine crawler, to the browser, all that's visible is an image.

SVGs on the Web

There's a better way. Earlier, the bulk of what you did with SVG was draw a yellow circle—not really exciting stuff. Now you'll see just how important it is to your craft.

Remember that SVGs are an XML-based vector graphic format. They support interactivity and animation. SVG is supported by nearly all major browsers and provides an incredibly lightweight way to visualize information.

SVGs can be built programmatically by a number of both server- and client-side languages. Later in the book, you'll be using a JavaScript library, D3.js, to build SVG visualizations. Each shape drawn is stored as an object on a scene graph in the DOM, which is then rendered as a bitmap. This means that if attributes of an SVG are changed, the browser can automatically re-render the element.

Native Desktop Browser Support

Since 2011, nearly all major desktop browsers have had some level of SVG support. Webkit browsers, such as Safari and Chrome, have had support for SVG1.1 since 2006. Browsers such as Firefox, which uses the Gecko engine for rendering, have at least partially supported SVG1.1 since 2005.

For a long time, the only holdout was Internet Explorer. IE8 and earlier do not natively support SVG. Internet Explorer 9 (released in 2011) was the first version to support the basic SVG feature set, with IE10 adding SVG1.1 filters. Vintage IE users have a wealth of plugins available to support SVG rendering, however.

Mobile Support

Smartphones, conversely, have terrific SVG support simply because their browsers were built in a time when vector support had already become standard. What's more, mobile browsers automatically update much better than older desktop browsers. When a browser keeps up-to-date whenever a new version is released, it's called "evergreen." Most mobile browsers, Chrome, Firefox, Safari, and the latest Windows Edge browser are all evergreen.

That's all there is to it. SVGs work well on mobile browsers. Hip hip hooray.

Meta Support

Don't forget about your screenless users, however.

Back in August of 2010, Google announced that it had begun to crawl and index SVG content on the web, and that users would begin seeing SVGs within search results. By February of 2011, they had announced that not only were SVGs crawled, and available in both web search and image search, but also users could limit their image searches by file type to *strictly* search SVGs. This is a massive boon for your visualization's very top layer: meta information.

Canvas

Canvas is an interesting alternative to SVG. It can also be drawn by code, and it can be output to the client, but it is actually raster-based, not vector-based. Canvas is

a lower-level API upon which an engine supporting SVG drawing *could* be built if not already present in the browser.

Whereas SVGs enable event handlers to be associated with objects (like other DOM elements), to achieve the same functionality with canvas you would manually have to match the coordinates of a click event with the coordinates of the drawn chart to determine whether it was clicked.

The bottom line? SVGs are the best solution.

Layering

Layering, in the context of visualization, is how you build your data so that it makes sense all the way from the raw level to the invisible meta information that you share along with it. Remember the progressive enhancement principles. Distilled, they are as follows:

- Basic content must be accessible to any browser.
- Semantic markup contains all web content.
- Layout and design are provided by externally linked CSS.
- Behavior enhancement is provided by unobtrusive JavaScript.

We can create an analogue to that, as we have already done implicitly in this chapter, for displaying data.

Raw and Processed Data

Raw data, or primary data, is the big set of unparsed data that you have at your fingertips. Raw data has not yet gone through any processing or manipulation. After you have begun to scrub (eliminate) unnecessary things,



NOTE

In the following chapter, Thinking Small, you'll learn some techniques for API-building that include how to effectively process your data.

you are left with processed data. What remains is just the data you will be presenting to your end users.

Processing your data is the first step in making it into something useful. You have stripped out what is either duplicate, outlying, or complicating entries, and started to focus on what will become useful in communication to end users.

API Response

The API response is your processed data shown as data. The examples you've looked at so far have been JSON, but more abstractly, they are a way for other machines to read the data that you are going to build markup around in the next layer.

API Input: Remember that whether you are planning to return your entire data set or only filtered responses, you will need to do so based upon your API's input. So far, you have only looked at the API's URI as an input technique, but this can become much more complicated when detailed requests between two different applications are being made.

API Output: Your output should have parity with what you will be displaying to users. Extra information should not be wasted on this layer, nor should additional data be pulled that is only visible to users with a screen.

Parsed Data

This is the layer of interpretation between the scrubbed data available from your API and the way you will be displaying it. In the train tracker example, it includes translating an arrival time of 0 to displaying "Arriving." In the flight predictions example, it creates the ratio number for your graph.

Markup

The markup is what will actually render in a browser. After doing whatever parsing is done to make the data into what is useful for a browser user (either visual, or

a web crawler), your markup is created to convey the same meaning as the API's matching end point. There are two layers to the markup:

Visible Markup: This is the actual HTML that creates your data visualization. The content that drives this markup is pulled from your API.

Meta Information: This content lives strictly on the markup layer, and it is used to provide screenless users with additional context, such as search engine crawlers to better understand what your data is describing when it can't be viewed.

Design

This is the data's visual styling, provided by your markup and CSS, and additionally affected by the implementation of your drawing technique. The analogue to web design would be CSS affecting how color, font size, margins, etc. would be displayed.

Interaction

This is the top-most layer. Unlike using JavaScript interactive effects in progressive enhancement, it can be thought of as progressive *exploration*. Interaction with certain elements of a data visualization can be paired with AJAX to pull in more data and draw further conclusions.

Responding to Data

Responding to the features of your data set is by far the most radical and important concept introduced in this chapter. For a moment, however, consider the actual data to be meaningless, and just some set called “Your Data.”

Just as you have abstracted away from devices, and from screens, let's abstract away from `your_data.json`. Just as you respond to features and constraints of devices, your data set also has more implicit features and constraints. They're not as obvious

as “a small screen” but will still impact how you can display information. Suppose your project brief has the following constraints:

- Most users are on mobile devices.
- There are seven different APIs to call.
- There is no guarantee of WiFi.
- Filtered information is missing for some subsets.

Clearly some of those are features of the users, and some are features of the data; but you are building to respond to both.

In its most basic sense, responsive web design consists of grids, media queries, and flexible media. What you have now added is flexible data.

Know Your Audience

It’s important to keep in mind that your data will be consumed by two sets of users, those with eyes and those without eyes. Although we tend to think of building visualizations for people, just as important is the user base that consumes your data invisibly and shares it with the rest of the world.

Build for Humans

To Consume: This is on the visible markup front. Obviously, you still want to make something efficient, concise, and beautiful. That’s why I put it first—it’s what you’ve been doing this entire time.

To Share: In your metadata, you’ll see soon that you can include social tags that convey just what you’re displaying. Semantic markup gets you a long way to begin with, but social networks are a screenless user themselves, as they then convey some snippet of your information to other screens, being shared as a teaser of your own site on someone’s news feed.

Build for Machines

To Consume: Everything starts with a good data set. Processing and scrubbing your data to make a clean, efficient, understandable API that other machine users can reach not only makes your life easier when transitioning into visual design of the data, but also enables other machine users to reach your API (if you choose) to consume your data and draw their own conclusions.

To Share: Semantic markup, meta tags, and schema.org structured markup are just a few of the techniques already covered. Every additional piece of useful information you can provide that gives someone who can't actually *see* your data a clearer idea of its content improves your visibility to search engine crawlers, and any other websites attempting to reach your data.

Know Your Data

As you reach the end of this chapter, there is one point that I want to hammer home: Know exactly what it is you have as data, and exactly what you're trying to convey.

By this point, you have seen numerous examples demonstrating how many unknowns you face as you attempt to create a responsive design. Your data, however, doesn't have to be an unknown. If you front-load the work to identify the exact nature of what raw data you are consuming, you can focus on optimizing for greater clarity.

Acquire meaningful data, scrub it, and present it in a way that is lightweight.

Summary

This chapter has focused on an important factor to which you must respond: data. It also emphasized a new class of user you must consider, non-human consumers of your data. In the next few chapters, you're going to take a path similar to the one you took with responsive design—getting elbow deep in some code examples to run through your toolset. Fortunately, at this point you should have a clear conceptual understanding of responsive data visualization as a unique subset of responsive design.

08

08

Thinking Small

Until now, the smallest screen you have considered has still been a screen. In this chapter, you're going to shrink the available real estate even further and begin your data visualization development at an even more basic level: the user without a screen. You'll learn to build API-first, rather than mobile-first, in order to create the most efficient, communicative representation of your data. From there, you'll be able to work with discrete, efficient data sets which you can group and layer to create effective visualizations.

Designing for the Smallest Canvas: No Canvas

Now that you have spent some time at the 40,000-foot view, in this chapter you will focus on some actual implementation details. Just as you have done in previous chapters, you should begin by considering the most limited user.

In the discussion about responsive design, that most limited user was someone working on a small screen on a bad network. And intuitively, that makes sense. You are designing visible layouts to be consumed by humans. The lower limit of that would, naturally, be something with which a human could interact.

Throughout this book, I've waxed poetic about the importance of developing for the smallest screen as your base user—and that's still a good precept to adhere to.

However, as brought up in the previous chapter, there is an important user who has something smaller than the smallest screen: a user with no screen at all. You see, below the *visible* layer presented to your human users is an absolutely pivotal layer to both your own and other systems: the API (application programming interface).

Your data is a big giant mess of numbers that probably will mean something to someone, but really means nothing to anyone, because you can't reach it (or can't reach it cleanly). Certain data sets are so large, or volatile, that attempting to dig through the



NOTE

Recall that data is collected from an operational environment. It is strictly raw findings and is perfectly concrete. Through processing and exploitation, data is distilled into information, which can be shared in a meaningful way. This information is analyzed, and produces some kind of output or findings, called knowledge.

actual data, or even keep it up to date, becomes more of a burden than extracting and presenting meaningful information.

An API should describe the expected inputs and standardized outputs of interacting with your data set.

The question is what makes a *good* API?

To answer that, it can be instructive to first come up with attributes that make a *bad* API. Don't worry if some of the terms in the next few paragraphs are unfamiliar; they are explained as the chapter moves along and you see examples of a good API.

Returning to the train tracker example you have been creating throughout this book, assume your current input is still a given train stop on a given line, and your output shows the next available trains in either direction, as shown in Figure 8-1.

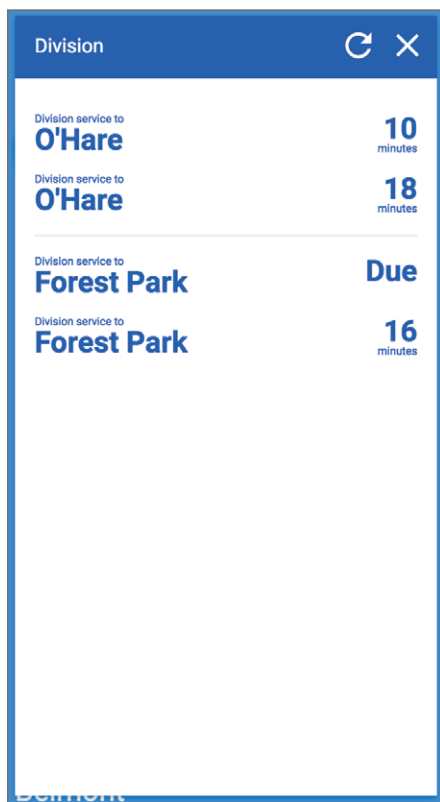


FIGURE 8-1

A train stop in your train tracker application

In previous chapters, you’ve built a nice, clean API that takes in and spits out the bare minimum information to build this application. In the real world, however, assume there is a level of security around this travel information, and that your first step is to get a key to the API.

An API key is a special string that allows you to interact with a private API. In this case, let’s say the owners of the API are monolithically “the government,” and their key request protocol is as follows: “E-mail `info@thegovernment.com` explaining why you want an API key.”

Therefore, you do that, and you wait. Meanwhile, paint dries.

When you finally get your API key, you have to search for documentation and examples of code that other people have written because no docs are provided with the API.

After you eventually get started, you notice that on top of requiring a key, the API returns only XML, and only allows for server-side requests; therefore, instead of just making a JSON request from the client on your server, you have to build out a PHP proxy on your server that reforms your request as a request to their server, pulls back the data, and then converts it to JSON for you.

Worse than that, it turns out that train station IDs aren’t always numbers, and they’re not unique to an individual train line. In addition, when a train station is under construction, the only indication is that all trains will resolve to an undetermined amount of wait time.

Even if some of that went over your head for the moment, you hopefully picked up on a few things that made the current implementation not-so-perfect. Now you’re going to build something much better.

Building for a Blind User

To get to an API, you first need some set of data that you actually want to reach. Although you’re working API-first, the API is never the actual *starting* point of your work. Posting open, bulk, structured data with good identifiers (more on this in a bit) has to happen before you could build a useful API.

An API itself is essentially Internet shorthand for program access to a system. Therefore, simply having a good API doesn't mean anything has been solved. Otherwise, you've just built a beautiful set of French doors, hoping that a building might be built around them someday.

Before you get to the visually designed and beautiful representation, you should take steps to make your data available to the public—whether that public is external users, machine users, or your own design team—in order to nail down just what your API should actually handle.

Open Data

The first step then is simply to expose your data. If you're willing to show it off when it's all dolled up on a Saturday, you should be willing to show it off all hung-over and crusty-eyed on a Sunday. You should be willing to expose your raw, un-filtered data to the public in order to see not only its benefits, but also its flaws.

Opening your data to a greater set of eyes exposes you to just how it is going to be consumed.

Bulk Data

Your data in bulk. Complicated, right?

This is simply an unfiltered dump of all the data available. What is useful about allowing access to bulk data is soliciting feedback regarding how users are attempting to interact (sort, filter, group, parse) with that data.

Structured Data

To have any meaning, your data should be structured in such a way that it is searchable, sortable, and machine-processable:

- **Searchable:** Can users search through the data and find something they are purposefully looking for?

- **Sortable:** Can related fields across items be compared and evaluated to a greater or lesser extent, allowing uniform items to be ranked by an individual field, or a join of multiple fields?
- **Machine-processable:** Without human intuition, can the data be scrubbed and analyzed? More simply, can a machine achieve the preceding two goals?

Structuring your data in a searchable, sortable, machine-processable way enables you to later more efficiently tailor your API to produce the exact output you're going to feed into your visualization, without any bloat or unnecessary bandwidth consumption.

If this seems familiar, it's because you have been building in a structured way from the beginning in order to give your designs meaning. Semantic markup is a way of structuring your websites.

In Chapter 7, I introduced the concept of `schema.org` structured tags—semantic markup for elements that aren't strictly analogous to print media. Whereas headings, paragraphs, footers, and asides are all supported tags in HTML, there obviously isn't a `<your-complicated-piece-of-data>` tag.

Good Identifiers

The identifying aspect of any one piece of data (the ID column, if you're thinking in terms of a table), is how that data is recognized as unique from another. To an API, this is still true at the database layer, but becomes more useful to a browser in the way of reaching a globally unique web address. What this means is that a RESTful web end point should universally point to the *same* data, regardless of what is trying to reach that end point.

Think of it this way: You have an email address that is uniquely yours. If someone emails you from their computer, their message should only come to you. Now, if they instead forward a list of cat gifs from a sharing app on their phone to that same email address, you would still expect that email to only reach you.

Intuitively, this is simply a 1:1 mapping of URL to data set.

A NOTE ON LINKED OPEN DATA

Combining the best parts of open, bulk, and structured data with global identifiers, and acting as a sort of API to combine data sets, linked open data (linkeddata.org) provides for a high level of interconnectivity between data sets in both the objects described and the concepts that relate said objects.

Linked open data is a concept backed by Tim Berners-Lee (remember when he created the web eight chapters ago?), and it is essentially an implementation of semantic web concepts within data sets. It is succinctly described on the linkeddata.org site as follows:

The web enables us to link related documents. Similarly it enables us to link related data. The term Linked Data refers to a set of best practices for publishing and connecting structured data on the web. Key technologies that support Linked Data are URIs (a generic means to identify entities or concepts in the world), HTTP (a simple yet universal mechanism for retrieving resources, or descriptions of resources), and RDF (a generic graph-based data model with which to structure and link data that describes things in the world).

Tom Heath (linkeddata.org)

We won't delve any further into linked open data, but you can win friends and influence people by mentioning the preceding quote.

The Good API

A web-based API cannot exist without a structured URL—a place where a browser (or a web-crawling bot) can actually get to the data. Your API isn't actually *creating* any new data, but is a way to publish all of your bulk data, and create more communicative end points for filters applied to that data.

If you're in a hurry, check out Waldo Janquith's instant-api PHP class (<https://github.com/waldoj/instant-api>), or the Python version (<https://github.com/jbradforddillon/instant-api-py>). It can automatically produce a class that spits out RESTful URLs for a set of static JSON data. For now, though, assume you're building from scratch, and want to make the API yourself.

The following sections represent the major tent poles of a good API. Many of these are discussed at length in other books, and I can't express enough gratitude for the smarts I distilled here from Joshua Tauberer (<https://opengovdata.io/>), and Ben Balter (<http://ben.balter.com/2013/12/30/data-user-experience/>).

Granular Access

If users want all of the data, they can download it. You've already created a bulk download, remember? An API isn't for that purpose—instead, it has to effectively provide users with access to the most granular level of your data.

For example, if your data was represented as a table, granularity means the capability to read a small slice of that table by filtering and sorting, or by directly hitting some unique identifier in your API. In Table 8-1, which shows a deliberately meaningless data set example, you should be able to easily select row A, column 3, and cell A3 with ease.

TABLE 8-1

Rows, columns, and cells should all be accessible uniquely.

	1	2	3
A	A1	A2	A3
B	B1	B2	B3
C	C1	C2	C3
D	D1	D2	D3

Just as important as what is returned is what should *not* be returned—that is, granular access to your data via your API means that anything outside of the requested outputs should *not* be returned, in order to reduce noise and bandwidth.

Deep Filters

If the granular item is the needle you're looking for in your haystack of data, filters are the magnets you use to find it. Searching through the full text of any data set would be ludicrous, and takes time, so an API should be able to handle either pre-populated (a drop down to the end user) or free-form text (a textbox to the end user) filters to narrow your searchable set.

Value Typing

Whenever possible, your response data should be typed values. That means that whenever a field's value is a string, an integer, a float, a date, or an object, it should be represented as such. In addition to that, types need to be *strictly* enforced: Your fields should uniformly pick a data type and stick to it. Note the following example:

```
{
  "stops": [
    {
      "LINE": "blue",
      "STATION": A,
      "STATION_ID": 123,
    },
    ...
    {
      "LINE": "blue",
      "STATION": B,
      "STATION_ID": "station_321",
    }
  ]
}
```

In the first block, you see that station A has a station ID of 123. However, station B has an ID of "station_321". JSON and XML with XSD are good at enforcing typing, as they are built in such a way that types are part of the language. CSVs or plain XML, however, are untyped, and data is much harder to parse. The following JavaScript shows the difference between == (double equal) and === (triple equal):

```
return 1 == '1'; //Not checking type

return 1 === '1'; //Checking type
```

In line 1, the code will resolve to true, even though int 1 and string “1” are of different types. However, line 3 compares both value and type, and will resolve to false.

In order to compare any sets of data meaningfully, you must do one of the following:

- Begin with typed data.
- Cast data to types.
- Build an interpreter layer above your API.

Normalizing and Denormalizing Data Tables

Normalization of a data set is the process of removing redundancy by separating the data into multiple sets. Primary keys that link tables should be used to reduce the amount of copied data, and instead provide references between items.

That said, making a query against a *denormalized* set is much faster than a *normalized* set. Intuitively, this makes sense: It’s faster to get data if it’s all returned in one big response than to make multiple chained API calls across multiple sets.

Re-denormalization (a ridiculous word) is good for querying only after a set has been normalized, as strictly denormalized data is bloated and difficult to maintain.

RESTful

The whole time you have been reading about good APIs, you have been looking at examples of REST, which stands for Representational State Transfer. REST is a set of practices that communicate over HTTP verbs (GET, POST, PUT, DELETE), which are used by browsers to retrieve web pages and send data to servers. Simply enough, just as every web page has a unique URL, every entry in your data set has its own URL as well.

You saw this in the train tracker example, where a URL to the list of stops on the blue line was

```
"/blue"
```

and a stop with ID 40890 in the collection of stops on the blue train line was as follows:

```
"/blue/40890"
```

The most awesome part of REST is that you already know how to speak it, because it's just a URL. The API becomes trivially simple to work with because it is both understandable and reusable as a hierarchical end point for all of your organized data.

Standardized (Selectable) Format

Recall from the earlier example of a bad API, you were able to retrieve the data only as an XML file, and only via a server-side call. You had to create a tiny PHP file on your own server that acted as a proxy to your server and which spat this same XML back, which you then immediately transformed into the JSON that you wanted.

That sucked.

Your API should provide a default output format, but also enable the user to select the output type, as a RESTful parameter in the URL. For example:

- `myapi.com/items/green.json` should return the JSON version of this end point.
- `myapi.com/items/green.xml` should return the identical data, but formatted as XML.

The most common outputs are JSON, XML, and CSV. CSV and XML are good for researchers, but not stellar for developers (due to lack of typing). JSON and XML with XSD are geared more for developers. Personally (along with a majority of the industry, it seems), I prefer to use JSON, as the format speaks nicely with the bulk of JavaScript that I use for data visualization and communication.

“Nice” Errors

Users will make mistakes. Servers will have latency. Someone will spill coffee on a laptop. Errors happen. Error messages, regardless of whether the problem is on the

client or the server, should provide a clear indication of exactly what is wrong, and (hopefully) a structured format explaining exactly what can be done to fix it.

In the following example, a request of `"/?station=foo"` was sent to the train tracker API. The response provides a clear message:

```
<ctatt>
  <tmst>20150617 23:33:47</tmst>
  <TimeStamp>20150617 23:33:47</TimeStamp>
  <errCd>103</errCd>
  <errNm>Invalid mapid: "foo"</errNm>
</ctatt>
```

You know exactly what went wrong and why.

The beauty of a RESTful API is that because it speaks through URLs on the web, it can also send back HTTP status codes, such as those in Table 8-2.

TABLE 8-2

Some meaningful HTTP status codes

Code	Description
200	Everything's OK. The request has succeeded.
400	The request could not be understood by the server due to malformed syntax. The client <i>should not</i> repeat the request without modifications.
403	Forbidden. The server understood the request but is refusing to fulfill it. Authorization will not help and the request <i>should not</i> be repeated.
404	Not found. The server has not found anything matching the Request-URI. No indication is given of whether the condition is temporary or permanent.
500	Internal server error. The server encountered an unexpected condition that prevented it from fulfilling the request.

Intents as URLs

Earlier in the chapter, I said that the first step in creating a useful data set, before building your API, is to share open bulk data and then see how it is consumed. This is where that comes into play.

As you see just how users sort, filter, and regularly interact with your data, pick the most important of these, and make them insanely simple to reach. As important as it is to create an API that is good at performing all filtering generically, with a bit of manual work (for machines), it is also important to absorb the complexity of these most common use cases and expose them specifically.

Looking at the data in Table 8-3, notice that each entry has a created date and an edited date.

TABLE 8-3

Data with comparable dates

ID	Name	Created Date	Edited Date
001	Foo	20150411	<i>null</i>
002	Bar	20140617	20150121
003	Foo bar	20150201	20150702
004	Foobar	20131128	<i>null</i>
005	Fubar	20150101	20150611

Suppose you want to find the last modified entry, to see whether it was an edit or a creation. It's certainly adequate to build a filter that finds the most recently created item by sorting, then build another that finds the most recently edited, and then join these and compare once again. However, why not instead create the following end point:

`myapi.com/items/latest`

This hides all of that implementation from a long RESTful end point and just returns the result of that work. Not only does this make the API much cleaner to a user, but it also grants you the flexibility to make backend changes invisibly later.

Meticulous Documentation

This should be more obvious than it is in the real world. An API without documentation is useless. It's a beautifully wordsmithed poem written in gibberish.

Documentation should cover why the data is important, what the fields mean, how to retrieve information using the API, and, most important, up-to-date examples. Whenever possible, make your documentation interactive, and allow more novice users to build their queries in real time.

Versioning

Versioning fits nicely with the preceding “nice” errors, as it is a way to prevent accidental mistakes when your API may change. Remember, at one point we all thought that Flash was going to be the way to create interactive data visualizations on the web forever.

Eventually things change, and your API is no different. Rather than force others (maybe even yourself) to update their client-side code to consume whatever change you have made to serve data, update your API’s version when a breaking change happens, but leave the old version working. This, too, can be built into the restful API, as something like the following:

```
myapi.com/v1.0/items  
myapi.com/v1.1/items
```

In a way, it’s progressive enhancement for your API consumers. Those who are able to work with the updated (and likely best) version can; if not, they are already using a working earlier version.

High Availability

You don’t have any clue when users will be attempting to reach your API. It could be perfectly spaced throughout the day and night (meaning that it needs 100 percent uptime at a relatively low scale), or it could be all at once during one hour of the day (meaning it needs 4.2 percent uptime at high scale). To be safe, you have to solve for both.

You can take basic precautions such as rate or query limiting to reduce the risk that your API comes under severe load. That said, you never want your API in a state in which users can’t get a full response within a reasonable amount of time.

Keep maintenance windows as short as possible!

High Performance

Your API should also be fast. Users will appreciate this because obviously they want it to be as fast as possible. There's not really any concept of *perceived* performance for an API: It is purely actual performance. Using the methods you've already learned about in this chapter, you should know how to prune your data, and provide granular access to keep queries efficient.

If an end point becomes too slow, you run the risk of impatient, dumb, or malicious users making resource-intensive queries that could ultimately bring your server down entirely. A complete discussion of API performance is beyond the scope of this book, but some important one-liners include the following:

- Don't allow for accidentally complicated queries by URL.
- Don't put resource-heavy authentication in the URL.
- Cache results on the server.
- Cache repeated queries on the client.

Open Development

Looking back at the example of a bad API, remember that the first step in using it was sending an e-mail to some mysterious government website and waiting some undisclosed amount of time.

Lame.

Forced authentication on your API is the easiest way to keep users from ever interacting with it. If you do happen to use an API key, make it an optional case for tiered users. Allow developers or designers within and outside of your team to quickly prototype against a public API. The only incentive for authentication is to allow a higher limit on queries.

This enables the greatest number of eyes (digital or human) on your data. These eyes will begin to conceptualize, develop, and share how they are consuming it; and as that happens, you will benefit from what they've shared and have the opportunity to react to their evolving needs.

Working this way allows for a more collaborative, developer/designer-centric ownership of your work. You'll learn more about the benefits of this mindset in the final chapter: Building the Future-Friendly Web.

The API-First Team

In theory, building API-first is all super-great stuff. It's bold to decide not to deliver anything that users can see as your first major release. The problem is...you're not going to deliver anything that users can see as your first major release.

If you're struggling to wrap your head around how this can benefit your team (or if you're trying to explain it to someone with dollar signs in their eyes), let's quickly walk through how this saves you time and effort later.

On the team level, your product creation milestones can look something like this:

1. Product vision development, user research:

- **Input:** Idea, data
- **Output:** User research and product benefits

2. Product discovery:

- **Input:** User research and general vision
- **Output:** Low fidelity prototypes of base experience. Core functionality outlined

3. Product inception:

- **Input:** Core functionality outline
- **Output:** API documentation

4. Functional teams break API down into stories:

- **Input:** API documentation and necessities for MVP
- **Output:** Backend team builds API. Frontend team builds UI with mock calls in parallel.

5. API changes discussed with product managers:

- **Input:** Working MVP
- **Output:** Specialized enhancements to product

As your team continues to build API-first, and you have established a stable base for all devices, you can progressively begin to hyperfocus on the features for which you want to optimize, in parallel with other teams.

This results in more natural-feeling interactions in a multi-platform environment simply because there is more time for engineering tasks to bake while the UI and user testing are happening.



NOTE

For more details on building API-first, visit api-first.com.

The Tiny Canvas

Now that you've spent some time working with no screen, you can dig into the case of a very small screen. A beautiful example of mobile data visualization is the Apple Watch. It's an impossibly small canvas, but it can present a wealth of information to a user by starting with a sensible default, allowing for layered interaction, and displaying granular data only when requested.

The Fitness App

Figure 8-2 shows the Apple Watch fitness app. With three concentric rings and some text, it certainly doesn't look like anything revolutionary. However, this small screen manages to convey a lot of information:

- Three different types of exercise are represented:
 - Stand is one of the types of activity.

- Stand is represented by the word “STAND.”
- Stand is represented by the color blue.
- Stand is represented by an arrow pointing up.
- The user has completed ~25% of his or her standing goal.
- Exercise is one of the types of activity.
- Exercise is represented by the word “EXERCISE.”
- Exercise is represented by the color green.
- Exercise is represented by two arrows pointing forward.
- The user has completed ~45% of his or her exercise goal.
- Move is one of the types of activity.
- Move is represented by the word “MOVE.”
- Move is represented by the color red.
- Move is represented by one arrow pointing forward.
- The user has completed ~55% of his or her movement goal.
- It is possible to display one thing at a time.
- This is view four of six possible views.

That’s an impressive amount of information being shown to the user, all rather elegantly. Apple chose this screen as the default because it provides, at a high level, both the status of the user’s goals and a language to understand the rest of the related views.

When a user interacts with one of the rings, it brings up one of the three possible views shown in Figure 8-3.

Now the visualization has changed to be more granular—it has been filtered down to an individual element of the greater whole. The language hasn’t changed—each activity type is represented by the same text, color, and icon—but now additional room is available to display more detailed information.



FIGURE 8-2

The Apple Watch fitness app's default screen



FIGURE 8-3

After interacting with any ring on the default screen (or swiping), one of these views is shown.

At this point, you can probably see how this is analogous to when you queried information about train lines when a user decided to interact with an element in the train tracker example. There is no benefit to cramming in exact numbers for all of the detailed fitness information because for the bulk of users, the most important use case is: “Have I gotten there yet?”

When users purposefully indicate that they want to know more, that not-yet-loaded, more relevant information can replace what is currently displayed.

Swiping in this view allows for even more detail (see Figure 8-4).



FIGURE 8-4

At an even more granular level, this view provides a timeline showing when activity took place throughout the day.

Here, the donut chart of “how much activity” versus “total activity goal” is removed. In its place is a timeline view showing points during the given day when activity has happened.

The colors are still present; and notice that because the visualizations are still at the level of detail considered “one type of activity,” their dots along the bottom remain uniform.

Again, the user has made it clear that he or she *wants* to see this information. It has been available all the time, but hidden beneath the more meaningful at-a-glance visualizations.

When working on such a small screen, this prioritization of *meaningful* information versus *all* of the information is a pivotal skill to learn.

Layering

The skill alluded to in the preceding section is one that I term *layering*. Layering is the prioritization of the most crucial and digestible information, while providing the means to obtain more granularity with user interaction.

Visual Layers

Remember from the first part of this book when you hid menus off-screen? Layering is a lot like that, except you hide all but one thing at a time off-screen, and you don't even bother loading it until it's requested.

Consider all the data available within that incredibly simple display of 1.5" × 1.5". On the very top layer, the data includes the following:

- Exercise metadata
- Exercise goal
- Exercise total
- Movement metadata
- Movement goal
- Movement total
- Stand metadata
- Stand goal
- Stand total

In addition, each type of exercise data includes at least the following:

- Metadata (name, icon, color)
- Goal total
- Current total
- Units
- 72 × 20-minute increments

The inclination to show all the granular data is certainly a strong one, but imagine the cost of returning $(72 + 4) * 3 = 228$ pieces of information instead of nine. Then imagine the insane challenge of trying to show that level of granularity.

Less Than Visual

Now consider the API driving this app for a moment. Plenty of other apps and web services have access to the data behind it through the HealthKit API. It's not quite what is created here, but the following is a simplified version:

End point 1: `/at-a-glance`

```
{
  "id":001,
  "name":"username",
  "activities" [
    "exercise": { ...exercise at-a-glance object... },
    "move": { ...movement at-a-glance object... },
    "stand": { ...stand at-a-glance object... }
  ]
}
```

This would likely call nested end points of 1a: `/[type]/at-a-glance`:

```
{
  "id":001,
  "type":"exercise",
  "goal": 30,
  "total": 16
}
```

End point(s) 2: `/[type]/ring`

```
{
  "id":002,
  "type":"move",
  "goal": 780,
  "total": 214,
  "unit": "cal"
}
```

End point(s) 3: `/[type]/timeline`

```
{
```

```
"id":003,  
"type":"stand",  
"total": 5,  
"unit": "hour",  
"increments": [  
  ...  
  "inc_1" : {  
    "amount": 1  
  },  
  "inc_2" : {  
    "amount": 0  
  },  
  ...  
]  
}
```

These of course are driven by the visualization shown. Because all of the data is simply available, combinations of the data (and a whole lot more not shown in this app) could easily be exposed.

More Than Visual

There's no reason to stop layering at the screen, though. In the case of the fitness app, voice notifications or device vibrations can interrupt you to let you know you should stand, or that you've met your exercise goal; these can then be augmented by quick interruptive layers on the screen displaying a simple message.

Visualization doesn't *strictly* have to stay visual.

Enhancing Efficiently

You've likely noticed that the chapter hasn't really said much yet about changing the visuals themselves as your user's device grows more advanced features.

Building out a clean, efficient API is the most pivotal part of responsive data visualization, as it satisfies the base condition of making the data universally available.

With granular access to a RESTful end point for your data, you could technically offload all the actual visualization work to whomever decides to consume it. (But don't, because you'll do it better anyway.)

Taking the preceding section of this chapter into account, now you can set the bar higher and create a visualization with the following:

- A 100%, A+, would-buy-again API
- A gorgeous, sensible default for the smallest screen

What comes next?

Growing Up

The solution that you would assume from responsive design techniques is to simply start showing more things as you have more screen space—and in many cases, this will likely work.

On a phone screen, that fitness app could likely fit the information hidden in each [type]/ring view on the default screen. On a tablet, at least one of the timelines could fit nicely.

However, you have to ask yourself, “Am I doing this just to do it?” There's a major difference between designing for text and images, and designing data visualization. With text and images, content has *all of* the focus. That is, users will be devoted to reading your text, or looking at your picture, or clicking your button.

In data visualization, however, there's a very dangerous tipping point at which too much information becomes Too Much Information. The goal is to communicate obvious conclusions from complicated data, so showing every layer of that complicated data can, in many cases, be just as bad as showing none of it. There is still a benefit to layering even when you have the real estate to show everything at once.

A simple way to do this, without oddly putting a 1.5" × 1.5" visualization in the middle of a 27" screen is to rely on your core responsive design skills. Jumping ahead a few chapters and assuming what you've built is backed by code, and not a

static image, you should be using totally scalable elements, with 100% relative sizes. As such, if you have screen space to grow into, grow into it.

Obviously, that has its drawbacks too, and continuing to follow the responsive design tenets, you can rely on end points to stop growing when something no longer makes sense at a comically huge size.

Recall from Chapter 7, though, that end points have a new meaning for you. You're not just solving for the amount of screen space available, but the amount of data. At a certain amount of real estate, the amount of discrete bars in the [type]/timeline view, for instance, could shift from 72 (every 20 minutes) to 288 (every 5 minutes). As long as the data is available, it's ready to be consumed at a cheap-to-create end point because you've built API-first.

First, Change Everything

Don't think of your visualization in a vacuum, though. Remember that it's still living within the context of a web page, dashboard, app, or holo-lens.

Changing the view of your data is one thing, but keeping it in the same amount of real estate always makes sense. The content around it is changing and reflowing responsively, so just by moving related or ancillary content closer you will have improved the visualization's clarity.

You actually went through an example of that when you built the cocktail guide mobile-first. Remember that an individual cocktail's page had the visualization in Figure 8-5 stacked above its content on a small screen.

However, as you had more real estate to consume, you pulled the content directly related to what the visualization represented next to it, making only minor adjustments to the size of the visualization itself (see Figures 8-6 and 8-7).



NOTE

As a double-secret special note, layering was actually used in that first example, because the information shown in the ingredient list was also made available when interacting with the corresponding slice of the donut graph.

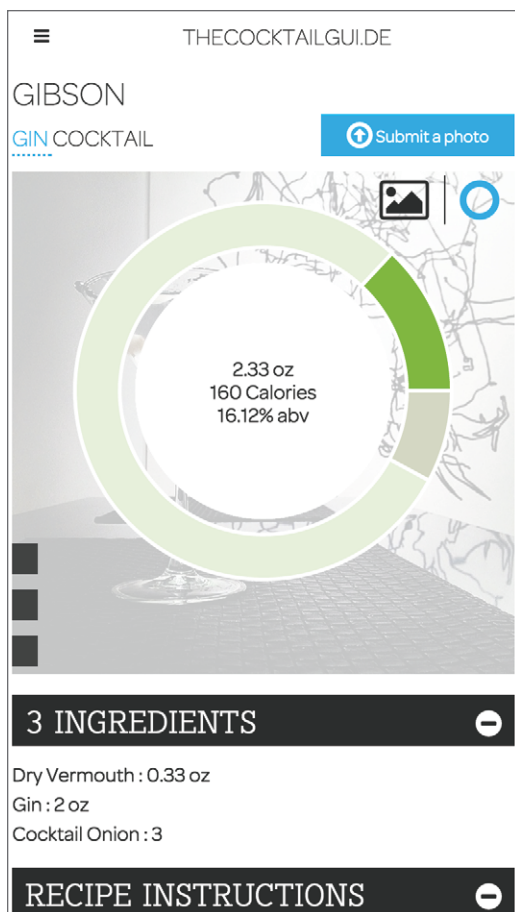


FIGURE 8-5

The cocktail guide's drink page, on a small screen

Responsive data visualization is not simply about causing your charts to constantly react to new features. Your goal is never just flashiness—it's clarity. As such, the design of your application as a whole can impact just how meaningful your visualization becomes.

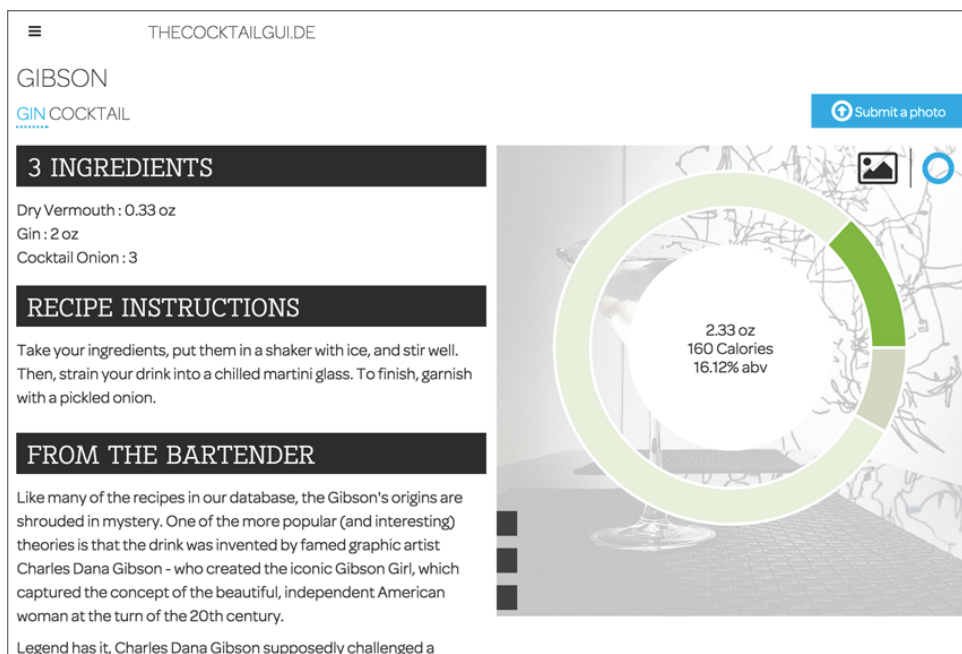


FIGURE 8-6

The cocktail guide on a medium screen. Notice the content shifting next to the visualization.

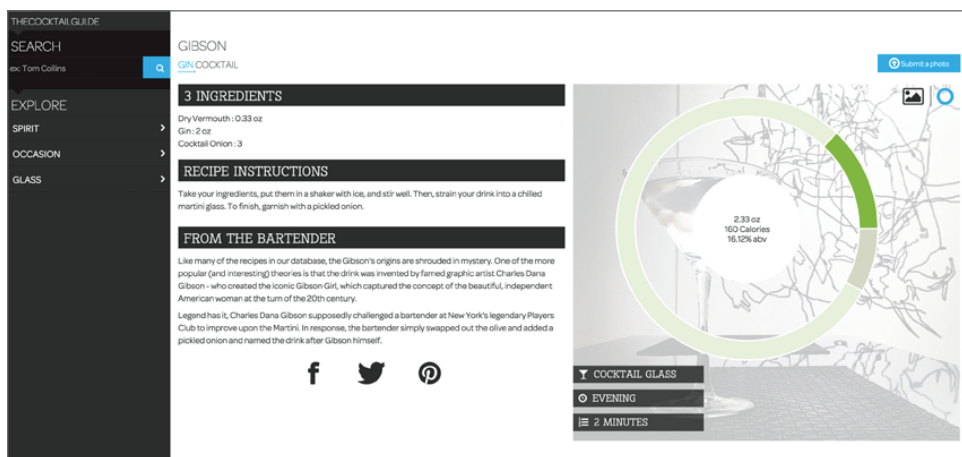


FIGURE 8-7

The cocktail guide on an even larger screen. There is very little change to the visualization, because making it any bigger would ultimately distract from its meaning.

They're Just Numbers

This chapter began with the concept of building for a headless user, a building approach called “API-first.” That’s great for this chapter, but how does that actually relate to being “mobile first” for all your users who can see and breathe and drop their phones in the toilet (again)?

Well...I just didn’t want to spoil it before: You’re building mobile-second.

That’s because the building mobile-first approach described thus far throughout this book has actually been API-first. Look back at what you learned before, and annotate exactly the steps you took: Simplify down to the very base, build semantically (structured data), and asynchronously pull in (which requires an API) anything that can be. Use additional subpages (which requires an API) in order to enable the retrieval of requested additional data (from your API), giving users a clean, bloat-free experience.

You’re already doing it by accident.

You’ve been building API-first and you didn’t even realize it. As I have emphasized many times in the preceding chapters, do your research first. Maybe your audience *is* truly 99% mobile, so you don’t need to worry too much about your API because you need to build an iOS app and be done. That’s great, and it should be your first goal.

That being said, designing for the next scale, and the next set of features, will become increasingly difficult as you attempt to wedge new features on top of old ones where they don’t fit.

I’ll now begin to refer to the mobile-first, responsive version of your data as the mobile *expression*. It is the most basic human-user-facing way in which you can convey your message. Unfortunately, binding your use cases to only this expression will inevitably hurt you down the road.

Before you are expressing anything, you are creating it; and as with everything in this book, your content must drive your design, and now your expression.

Summary

Building API-first is the key to creating responsive data visualization. You now have a grasp on the lowest level of functionality that you need in order to make your data accessible. You've learned how to create an efficient, effective API to reach your data, and how to structure that data. Additionally, you've seen how that data can begin to back visualizations on the smallest of screens.

09

09

Asset Dependence

This chapter is the last piece of radically new information presented in this book before you start merging everything you've learned in the preceding chapters into a working whole. In this chapter, you are going to look at how JavaScript's D3 library works, and learn how to react not to client feature changes, but rather to changes in the data backing your visualizations.

Dynamic Data

This is your first big foray into data-driven design. In the previous chapter, you were working toward building for the smallest screen (no screen), and then creating a basic experience. That experience has already begun to meet the classic definition of responsive design—that is, responding to the client’s browser.

However, having read and absorbed all the responsive data visualization tenets, you also know that ensuring client responsiveness is only half of your job when it comes to building data visualization. The other half is responding to your data set.

You are already closer to mastering this than you might think. After all, you’ve learned about building an API that is dynamic enough to spit out exactly the level of granularity you want. You know how to react to data changes by redrawing your chart (a refresher and much more on that in a few pages). You have a grasp of grouping data. And in the previous chapter, you worked through an example of data layering.

Pretty good...but now it’s time to get serious.

Dipping Your Toe In

In this section you’ll begin building visually with data. You’ll do that with a JavaScript library called D3.js (data-driven documents). If you have never used it, don’t fret—the next chapter starts with the assumption that you are first hearing about it in this book.

In this chapter, you’ll examine a few pivotal parts of D3 that make it so attractive for working with responsive data visualization.

D3.JS

D3.js, simply put, is a library of JavaScript functions for manipulating HTML documents based on data. It enables you to pump in big sets of (clean, structured) data, and spit out CSS style-able scalable vector graphics (SVGs).

Because it isn't a plugin or a third-party system, such as a Flash container or a Java applet, D3 enables you to use all the functionalities of modern browsers that you learned to harness in the first half of this book.

If you are familiar with jQuery (which, come on, you are), you know that its core value is the capability to easily manipulate the DOM with JavaScript. In fact, the syntax of D3 reads in a syntactically similar way to jQuery. With D3, you are simply manipulating the DOM with JavaScript *and data*.

It allows you to *bind* data to the DOM and then apply transformations to the document based upon that data. Suppose, for example, you have an array of the following:

```
[0, 2, 8, 5, 1, 9]
```

You could use D3 to create an HTML table that displays only those numbers in a row. That is a totally valid use case.

You could *also* use D3 to create an interactive bar chart that transitions in from zero-height and paints each bar with a hexadecimal color value based upon some math done on the length of the bar. That is also a valid use case.

In other words, you're not bound to a specific type of visualization, or tied to any one implementation. The purpose of D3 is to provide an efficient, simple bridge between data and the DOM.

JSON to HTML

Since it is simply a JavaScript library, D3 does not create any amazing new way to represent data. The elements available to you as a designer are exactly what you're already familiar with: HTML and CSS. Check out the following simple example, the syntax of which should be easy enough to parse:

```
d3.selectAll('div.change-me').style('height', function() {  
  return Math.random()*100+'px';  
});
```

The preceding code finds the `div`s with class `.change-me` and gives them a random height. Although D3 was used for this example, it wasn't necessary. The following demonstrates an example with data:

```
d3.selectAll('div.change-me')
  .data([0,2,8,5,1,9])
  .style('margin-bottom', function(d) { return d + 'px'; });
```

The pivotal point here is that you're now beginning to use D3 as a bridge between your structured data in JSON and the outputted HTML representation of that data. That is, on the input side you will always have JSON, which is itself structured (and machine-readable) and on the output side, you will have some visualized HTML representation of that exact same data.

Data()

Notice in the preceding code the use of what appears to be a function called `data`. In D3, this is called a *selection*, and it does exactly what you would expect—it takes in data.

What exactly it does with this data is more interesting. The preceding example passed in an array of numerical values, and then within the function referenced a variable `d`.

That variable actually represents each one of the datum from the set. You can think of it as an iterator across all of the *selected* elements, receiving each `d` from the `data`. In plain English, the code means the following: For all of the selected divs with class `.change-me`, iterate through the data elements and bind each one with a corresponding div. Then, on that div, add a style of that data's value as the div's bottom margin.

Enter() and Exit()

You'll learn more about the implementation and importance of D3's data join functions (and also what a data join function is) in the next chapter. For now, know that D3 has two selections called `enter` and `exit`.

Using these, you can create new DOM nodes for incoming data, and remove nodes for data that is no longer part of the set that you want to display.

When you *bind* data to a selection, each element of data in the inputted object is paired with a corresponding DOM node. When you reach a point at which there

is more data than nodes, the `enter` selection instantiates the necessary additional nodes. You can see this in the following example:

```
d3.selectAll('div.change-me')
  .data([0,2,8,5,1,9])
  .enter().append('div.change-me')
  .style('margin-bottom', function(d) { return d + 'px'; });
```

`Exit`, as you might expect, does the exact opposite. When a situation arises where data has been removed from the set but still has corresponding nodes in the DOM, you can call the `exit` selection to handle those nodes. Check out the following simple `exit`:

```
var changeMe = d3.selectAll('div.change-me')
  .data([0,2,8,5,1,9]);

changeMe.enter().append('div.change-me')
  .style('margin-bottom', function(d) { return d + 'px'; });

changeMe.exit().remove();
```

The key thing to notice is that instead of simply chaining `data`, `enter`, and `exit`, I set a variable of `changeMe` to handle the data. This way, all three selections can be handled separately and efficiently.

That's all the detail you need at this point, although it's worth noting that the default selection for D3 is simply `data`, not `enter` or `exit`. Therefore, not using `enter` and `exit` only selects the elements for which there exists both data and DOM nodes.

For your own purposes, splitting `data`, `enter`, and `exit` will prove beneficial as it enables you to work separately on the following tasks:

- Updating existing nodes
- Adding new nodes
- Removing unwanted nodes

If you abstract the selections out a bit, you can see how they are related to a few of the responsive design tenets. Updating existing elements happens within media queries, and adding and/or removing elements corresponds to what happens at your end points.

Data versus Screen

With a few tools from D3 under your belt, and the ability to follow some code logic in JavaScript, I'm going to throw a trivial graph at you. The interesting thing about this example is that it doesn't reflect a static data set. Assume you have the following array of JSON data backing your visualization:

```
json_data = [  
  { 'id': 0, 'created_at': 'May 01 2015 00:00:00', 'val': 100},  
  { 'id': 1, 'created_at': 'Jun 15 2015 00:00:00', 'val': 300},  
  { 'id': 2, 'created_at': 'Jun 01 2015 00:00:00', 'val': 900},  
  { 'id': 3, 'created_at': 'May 15 2015 00:00:00', 'val': 2700},  
  { 'id': 4, 'created_at': 'Jul 01 2015 00:00:00', 'val': 8100}  
]
```

You are now in a world where responding to data is the most important responsive design goal (at least for a few pages). The graph of this data will be simplified enough in order to *visualize how* enter *and* exit *enable you to manage data responsiveness*.

Assume the DOM starts out looking like this:

```
<div id='viz'></div>
```

The D3-based JavaScript in its entirety is shown here:

```
(function() {  
  var data = json_data.slice()  
  var format = d3.time.format('%b %d %Y')  
  var computeVal = function(d) { return d.val }  
  var computeDate = function(d) { return format.parse(d.created_at) }  
  
  var x = d3.time.scale()  
    .range([8, 304])  
    .domain(d3.extent(data, computeDate))  
  
  var y = d3.scale.linear()  
    .range([144, 8])  
    .domain(d3.extent(data, computeVal))  
  
  var svg = d3.select('#viz').append('svg:svg')  
    .attr('width', 320)  
    .attr('height', 160)  
  
  svg.selectAll('circle').data(data).enter()
```

```

.append('svg:circle')
.attr('r', 4)
.attr('cx', function(d) { return x(computeDate(d)) })
.attr('cy', function(d) { return y(computeVal(d)) })
}) ();

```

You've learned already that just creating the SVG will result in the DOM shown in Figure 9-1.

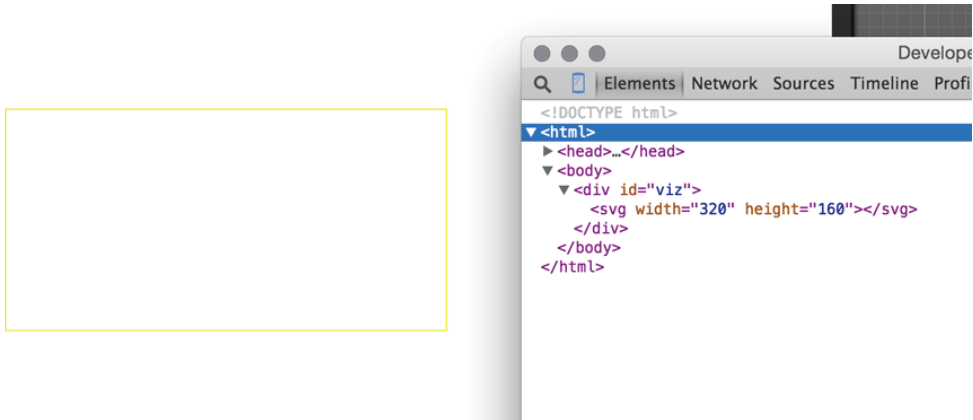


FIGURE 9-1

The outputted DOM from the D3 SVG insertion

Figure 9-2 shows the DOM with the values populated from your JSON data.

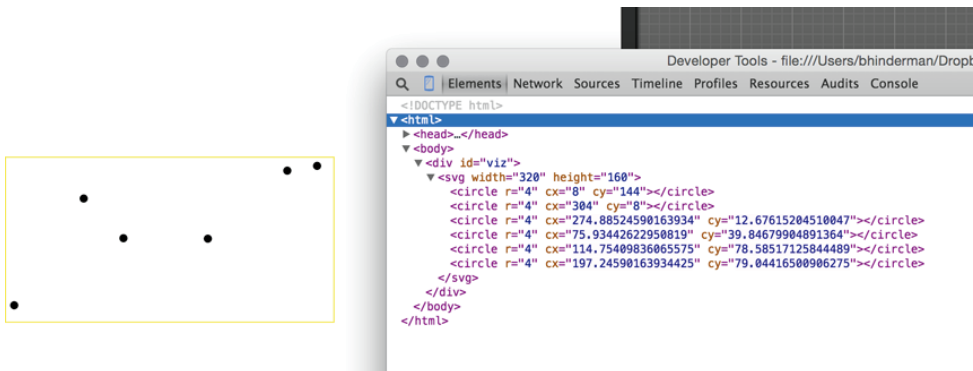


FIGURE 9-2

The DOM created by your data-bound D3 code

Determining Domain

The domain of your data set is the start and end points. It can be any sort of value that can be compared in JavaScript. In this example, one domain is a set of numbers, while the other is a set of dates. You can do whatever math you would like (or hard-code numbers) to set a domain, but D3 provides a nice function to compare the data shown in the following code:

```
d3.extent(data, computeDate)
```

This acts as a wrapper function around the following:

```
[d3.min(data, dateFn), d3.max(data, dateFn)]
```

The domain of your visualization responds to changes in your data set, while it remains uniform when the client's screen capabilities change.

Determining Range

The range of your data set is the pixel size (just pretend you're not going to change that later) in the DOM. Before making your visualization responsive, you'll likely want to cover the entire width and height of your SVG's canvas.

In the current example, I'm using an arbitrary 8px padding on all sides of my D3 visualization in order to keep any elements from clipping the edge of my SVG. The relevant pieces in the preceding code are called out here:

```
...
var x = d3.time.scale()
    .range([8, 304])
...
var y = d3.scale.linear()
    .range([144, 8])
...
var svg = d3.select('#viz').append('svg:svg')
    .attr('width', 320)
    .attr('height', 160)
...
```

At this point, the pixel values don't mean anything yet.

More often than not, the range of your visualization will respond to the client's screen size and capabilities, and will remain uniform in response to data changes.

Reacting to Data Changes

Now that you have a basic chart, it's time to react to data changes. You'll start with adding data, as it is substantially easier than removing or modifying existing elements.

Remember from earlier in the chapter that `data`, `enter`, and `exit` each bind datum to DOM nodes. *How* they bind them is, by default, dependent upon `data`. Suppose you have the following code:

```
svg.selectAll('circle').data(data).enter()
  .append('svg:circle')
  .attr('r', 4)
  .attr('cx', function(d) { return x(computeDate(d)) })
  .attr('cy', function(d) { return y(computeVal(d)) })
```

Once your data is bound to the selection in lines 2 and 3, `enter` will return the values for which no `.data-rect` is found. The change that needs to be made is binding the new datum to a new node, and attaching it to the DOM.

Adding Data

The good news is that your current code very nearly supports dynamic data addition. Assume the following function that waits every five seconds to push a new random element to the set:

```
var addInterval = setInterval(function() {
  var start = d3.min(data, computeDate);
  var end = d3.max(data, computeDate);
  var time = start.getTime() + Math.random() * (end.getTime() - start.getTime());
  var date = new Date(time);

  var newElement = {
    'id': Math.round(Math.random() * 10),
    'val': 100 + (3 * (Math.random() * 2666)),
    'created_at': date.toDateString()
  }
}, 5000);
```

(continues)

(continued)

```
};  
//data is still json_data from above  
data.push(newElement);  
,5000);
```

Just adding elements to the set won't actually do anything. The DOM is not actively listening to changes to the data set, and your nodes are only those originally added in the `data` selection.

Therefore, first assume there is a function `redraw()` that updates the visualization. It would be placed like this:

```
var addInterval = setInterval(function() {  
    var start = d3.min(data, computeDate);  
    var end = d3.max(data, computeDate);  
    var time = start.getTime() + Math.random() * (end.getTime() - start.  
getTime());  
    var date = new Date(time);  
  
    var newElement = {  
        'id': Math.round(Math.random() * 10),  
        'val': 100 + (3 * (Math.random() * 2666)),  
        'created_at': date.toDateString()  
    };  
    //data is still json_data from above  
    data.push(newElement);  
    redraw();  
,5000);
```

Notice that with the preceding code, any new element will still fit within your domain. No responsive redrawing needs to happen. What happens within `redraw` is now the enter selection:

```
var redraw = function() {  
    svg.selectAll('circle').data(data).enter()  
        .append('svg:circle')  
        .attr('r', 4)  
        .attr('cx', function(d) { return x(computeDate(d)) })  
        .attr('cy', function(d) { return y(computeVal(d)) })  
}
```


The `redraw` function simply acts as a wrapper around the `enter` selector, and handles what you do each time an unbound element is found. The entirety of your code now looks like this:

```
(function() {
  var data = json_data.slice();
  var format = d3.time.format('%a %b %d %Y');
  var computeVal = function(d) { return d.val };
  var computeDate = function(d) { return format.parse(d.created_at) };

  var x = d3.time.scale()
    .range([8, 304])
    .domain(d3.extent(data, computeDate));

  var y = d3.scale.linear()
    .range([144, 8])
    .domain(d3.extent(data, computeVal));

  var svg = d3.select('#viz').append('svg:svg')
    .attr('width', 320)
    .attr('height', 160);

  var redraw = function() {
    svg.selectAll('circle').data(data).enter()
      .append('svg:circle')
      .attr('r', 4)
      .attr('cx', function(d) { return x(computeDate(d)) })
      .attr('cy', function(d) { return y(computeVal(d)) })
  };

  var addInterval = setInterval(function() {
    var start = d3.min(data, computeDate);
    var end = d3.max(data, computeDate);
    var time = start.getTime() + Math.random() * (end.getTime() -
start.getTime());
    var date = new Date(time);

    var newElement = {
      'id': Math.round(Math.random() * 10),
      'val': 100 + (3 * (Math.random() * 2666)),
      'created_at': date.toDateString()
    };
    data.push(newElement);
    redraw();
  }, 5000);
```

(continues)

(continued)

```
    redraw();  
  })();
```

Outside of Your Domain

Here's where it starts to get tricky. What happens if your next added element's `val` is 8200, or `created_at` is July 16?

Remember that in order to build a graph in D3, you have to supply both a domain (max and min data values) and range (max and min pixel values), but 8200 is outside of the current graph's range of 100–8100.

I'll refer to the domain and range now as the chart's *scale*. In order to append this new value to the chart, you need to redetermine its scale. The three steps to doing this are as follows:

1. Update the domain with the new data set, so that the new node fits.
2. Add the new node.
3. Update existing nodes with the new scale, so that they move to the correct position on the rescaled chart.

To do this, the `redraw` function needs to take into account the following two steps: add the new set, and then adjust the full set. Consider this code:

```
var redraw = function() {  
  x.domain(d3.extent(data, computeDate));  
  y.domain(d3.extent(data, computeVal));  
  
  var items = svg.selectAll('circle').data(data);  
  
  items  
    .attr('cx', function(d) { return x(computeDate(d)) })  
    .attr('cy', function(d) { return y(computeVal(d)) });  
  
  items.enter()  
    .append('svg:circle')  
    .attr('r', 4)  
    .attr('cx', function(d) { return x(computeDate(d)) })  
    .attr('cy', function(d) { return y(computeVal(d)) });  
};
```

As shown in lines 2 and 3, the domain is being reset based upon the updated data set. Because the screen size hasn't changed at all, and you are only reacting to a data change, the range can remain. The only thing that needed to change was the domain. Item one on your list is done.

What's going on in the rest of that code, though?

Moving and Shaking

The result of the `data` selector has been set to a variable: `items`. This is because `enter` affects only unbound (new) nodes; therefore, it's simply inefficient to check `data` again. Remember from earlier that you handle adding, removing, and updating elements uniquely for efficiency and readability's sake.

The remainder of the code is some D3-powered math to do...something. The gist is as follows: When you add a new node, the already rendered nodes do not receive an event that this happened. Adding nodes, changing domain or range, the DOM doesn't care, and the nodes will simply fall out of sync with your changes.

Therefore, upon refresh, the `cx` and `cy` of each existing element in `items` must be redrawn to the new scale. In this example, they are just moving, but hopefully you can already see the analog to changing height, width, or any other style-able attribute when responding to changes in data.

Upon refresh, the chart now looks like Figure 9-3.

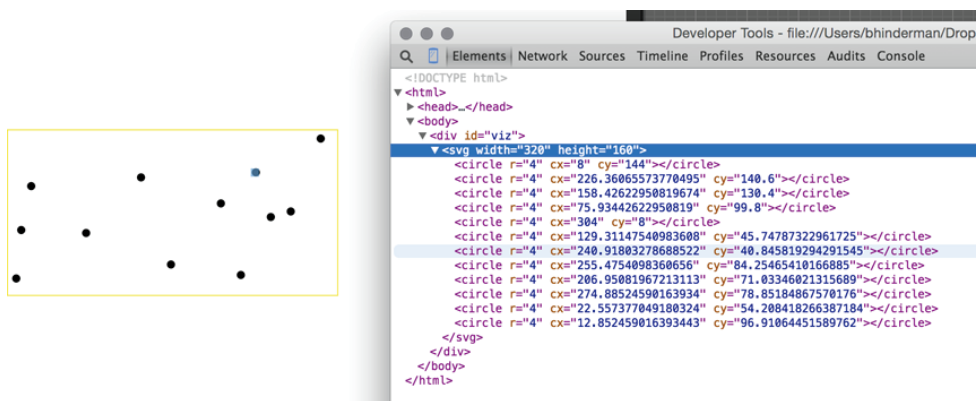


FIGURE 9-3

The DOM after many elements have been added

Removing Data

Naturally, wherever you add things, you'll also want to remove them. That's going to be a problem at the moment. Suppose there is currently a function that removes a random element every six seconds, as shown here:

```
var removeInterval = setInterval(function() {  
    var item = Math.floor(Math.random()*data.length-1);  
  
    //data is still json_data from above  
    data.splice(item,1);  
    redraw();  
},6000);
```

The problem is that the current data selection does not have any function to reference the data as a key-value pair. It's simply looking at an array and binding it mindlessly to some DOM elements:

```
var items = svg.selectAll('circle').data(data);
```

Previously, you handled data for the chart by adding it inline once, which does not save any key-value. Thankfully, creating a key for a node-element binding is not nearly as difficult as you might think. Consider the following code:

```
var items = svg.selectAll('circle').data(data,computeDate);
```

The difference here is that the elements now have an assigned key—the `created_at` that is taken from the `computeDate` function. Not only does that enable you to remove elements, it also solves a problem that perhaps you thought of at the beginning of this example: resolving duplicate nodes.

In addition to the data change, you can now add the `exit` selector to handle removing elements. That code is simply as follows:

```
items.exit()  
    .remove();
```

With this code, you can see what happens to the DOM after some time in Figure 9-4.

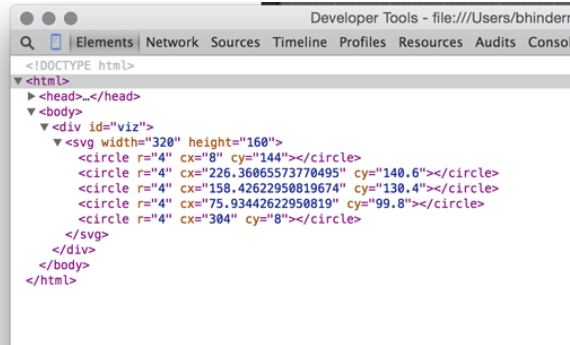
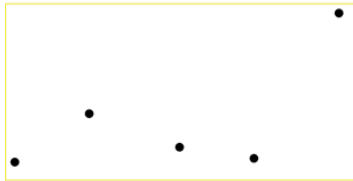


FIGURE 9-4

The DOM after many elements have been added and removed

It works this easily, for now, because each node in `items` is now uniquely identifiable. Just for fun, the entirety of your code (without the code to add a node every five seconds) is shown here:

```
(function() {
  var data = json_data.slice();
  var format = d3.time.format('%a %b %d %Y');
  var computeVal = function(d) { return d.val };
  var computeDate = function(d) { return format.parse(d.created_at) };

  var x = d3.time.scale()
    .range([8, 304])
    .domain(d3.extent(data, computeDate));

  var y = d3.scale.linear()
    .range([144, 8])
    .domain(d3.extent(data, computeVal));

  var svg = d3.select('#viz').append('svg:svg')
    .attr('width', 320)
    .attr('height', 160);

  var redraw = function() {
    x.domain(d3.extent(data, computeDate));
    y.domain(d3.extent(data, computeVal));
```

(continues)

(continued)

```
var items = svg.selectAll('circle').data(data);

items
  .attr('cx', function(d) { return x(computeDate(d)) })
  .attr('cy', function(d) { return y(computeVal(d)) });

items.enter()
  .append('svg:circle')
  .attr('r', 4)
  .attr('cx', function(d) { return x(computeDate(d)) })
  .attr('cy', function(d) { return y(computeVal(d)) });

items.exit()
  .remove();
};
redraw();
})();
```

And you're done! This chart can now populate, add, remove, and redraw nodes to scale correctly. You have used the extent of the data to automatically keep up to date with the minimum and maximum of your domain, and have applied a key function to create primary keys and de-duplicate your bound nodes.



NOTE

This code is also all available in the Chapter 9 section of responsivedata.visualization.com.

Tying Visualization to Screens

That's all well and good, but you might have noticed that I didn't start this chapter with a real use case. In fact, it was emphasized that the example uses an *arbitrary*

set of data. How, then, does reacting to changes in data actually apply to real-world responsive design?

I'm so glad you asked! In previous chapters, you learned that data should be *grouped* and *layered*. Grouping is used in response to a disconnect between the available space for visualization and the complexity of the data set. Layering is an outgrowth of using reasonable defaults in responsive web design to provide the most useful view immediately, adding levels of granularity when requested.

If you connect these concepts to what you just learned about responding to data changes, you can see that you're creating different *expressions* of your data. Regardless of scenario, you will need to optimize for certain levels of either data granularity or feature availability. Combining grouping, layering, and responsive redrawing enables you to meet those requirements.

Grouping Again

From the list of responsive data visualization tenets, remember that grouping is a technique for getting more meaningful insights about data when it is any of the following:

- Too big
- Too complicated
- Too small

It has a close analog to the idea of a breakpoint, but whereas the breakpoint is conceptually the point at which the entire chart should be redrawn, the grouping point is one at which certain related items in the data should be shown as one item (or split back apart).

Consider the `redraw` function from the example. You called it whenever an element had been added or removed from the data set, and you needed to do the following:

1. Update the domain with the new data set, so that the new node fits.

2. Add the new node.
3. Update existing nodes with the new scale, so that they move to the correct position on the rescaled chart.

However, nothing prevents you from extending it beyond simply adding/removing. Sometimes (such as when your screen width doubles), you may have the opportunity to pull back data from a more granular API end point. Remember the following code from Chapter 7:

```
$document.ready(draw);
$(window).resize(draw);

function draw() {
  var w = $(window).width();
  var h = $(window).height();

  // Call re-draw functions
  if(w > 768) {
    unDraw();
    draw768();
  }
  ...
  else {
    unDraw();
    drawDefault();
  }
}
```

In that code, you listen for the page's size, then call an individual function that then undraws something and redraws it based upon the new size.

Combine that with your basic knowledge of D3's selector methods now, and you can begin to see what it would look like if you were able to do the following, and in this order:

1. Check for a resize.
2. Change your data set based upon the resize.
3. Redraw your chart with the new data.

Suppose you have the function `getData`, which takes in a parameter of `granularity`, and another function `getGranularity`, which provides the granularity of the current data set. Using the same logical viewing modes that were defined in the preceding code, you can abstract your workflow to the following pseudocode:

```
function redraw() {
  var width = window.width();
  var currentGranularity = data.getGranularity() || 'large';
  if(width > 960 && currentGranularity !== 'small') {
    data.setGranularity = 'small';
    data.empty();
    data.push(getData('small'));
  } else if(width > 680 && currentGranularity !== 'med') {
    data.setGranularity = 'med';
    data.empty();
    data.push(getData('med'));
  } else if(currentGranularity !== 'large') {
    data.setGranularity = 'large';
    data.empty();
    data.push(getData('large'));
  }
}

redraw();
window.on(resize) { redraw(); }
```

Here, you can see that the default case is set to a large granularity, as this would presumably be the cheapest information to fetch. Within the `redraw` function, different levels of granularity are requested at different screen widths.

The function itself is run as soon as the page is loaded, and then again on every page resize.

This code assumes that you have created `enter` and `exit` selectors in order to handle each time you `empty` out of and `push` into your data set.

Summary

Whether you realize it or not, you now have every piece you need to start building visualizations responsively. You know how to change markup in response to client features, you know how to write data as markup, and you know how to change that markup when the data changes.

In the next chapter, you're going to dig deeply into D3, combining everything you have learned so far. It's time to start putting all of these pieces together.

Try It

Head to responsivedatavisualization.com. In the section for this chapter, you'll find both the code for the responsive scatter plot you created in this chapter and a few challenges:

- For the first challenge I've created a jQuery function that simply adds a group of elements to a page, based upon values in a JSON object. Re-create this functionality using D3 instead of jQuery.
- For the second challenge I've partially completed a bar chart that renders correctly using D3. It has some additional functionality to add and remove elements from the data set, but it is missing handlers for `enter` and `exit`. Your task is to implement that functionality.

10

10

Code-Driven Visualization

Throughout this book, you have been moving steadily toward a code-driven, device-independent way of designing and thinking. On the visualization side, you have even built for users without screens. Now it's time to bring together your knowledge of building resolution-agnostic web experiences with your understanding of clean, granular API construction in order to create lightweight, responsive data visualizations.

Unknown Inputs and Outputs

Four tenets have driven each step of your responsive design thus far:

- Be universal
- Be flexible
- Be economical
- Be specific

Those four principles had slightly different meanings when speaking to web design (Chapter 2) versus data visualization (Chapter 7). In dealing with both, you have had to come up with solutions to respond to (at the very least) the following:

- Screen size
- Screen orientation
- DPI
- Network speed
- Bandwidth caps
- Media type
- Content flow
- Browser support
- Touch and cursor
- Asset size
- Data specificity
- API granularity
- Data size
- Data complexity
- Data cleanliness

If you look closely at that long list of specific items, however, you can see that they fall under two distinct categories in terms of responding to a design:

- The user
- The data

In other words, every single thing that you can do to make your visualization more clear, more usable, or more communicative is a reaction to either the context in which the user is trying to consume it, or a change in the data being displayed.

In every situation, responding has *not* meant assuming that your design is going to be a pixel-perfect match of some Illustrator mockup. Instead, you have seen the need to adapt, resize, and reflow your design in order to best fit within a multitude of limitations.

That is, you have relied less on static assets, and more on dynamic code-driven solutions. Before moving forward, take a moment to review just how you've responded to each of these two unknowns.

To the User

Responding to the user's context is the core of not only responsive data visualization, but responsive web design as a whole. That's why so much of this book isn't strictly centered on data. If you think of your visualization as a simple container, as shown in Figure 10-1, then there is no reason to treat it any different from a paragraph of text, or an image.

In this sense, you can think of *designing* for the reflow of a visualization in the same way that you would for any other content. It needs a certain amount of space, and it lives within a certain container. The difference is how you implement your responsive techniques.

To the Data

Responding to the data is an unknown totally unique to data visualization, and that's what makes it such a more complex topic. There is no concept of a blog post's content changing because a user decided to filter down to only words that begin with the letter W. Instead of Figure 10-1, the situation here is related only to the things happening *within* the visualization, as shown in Figure 10-2.



FIGURE 10-1

A data visualization is just another block of content on a page.

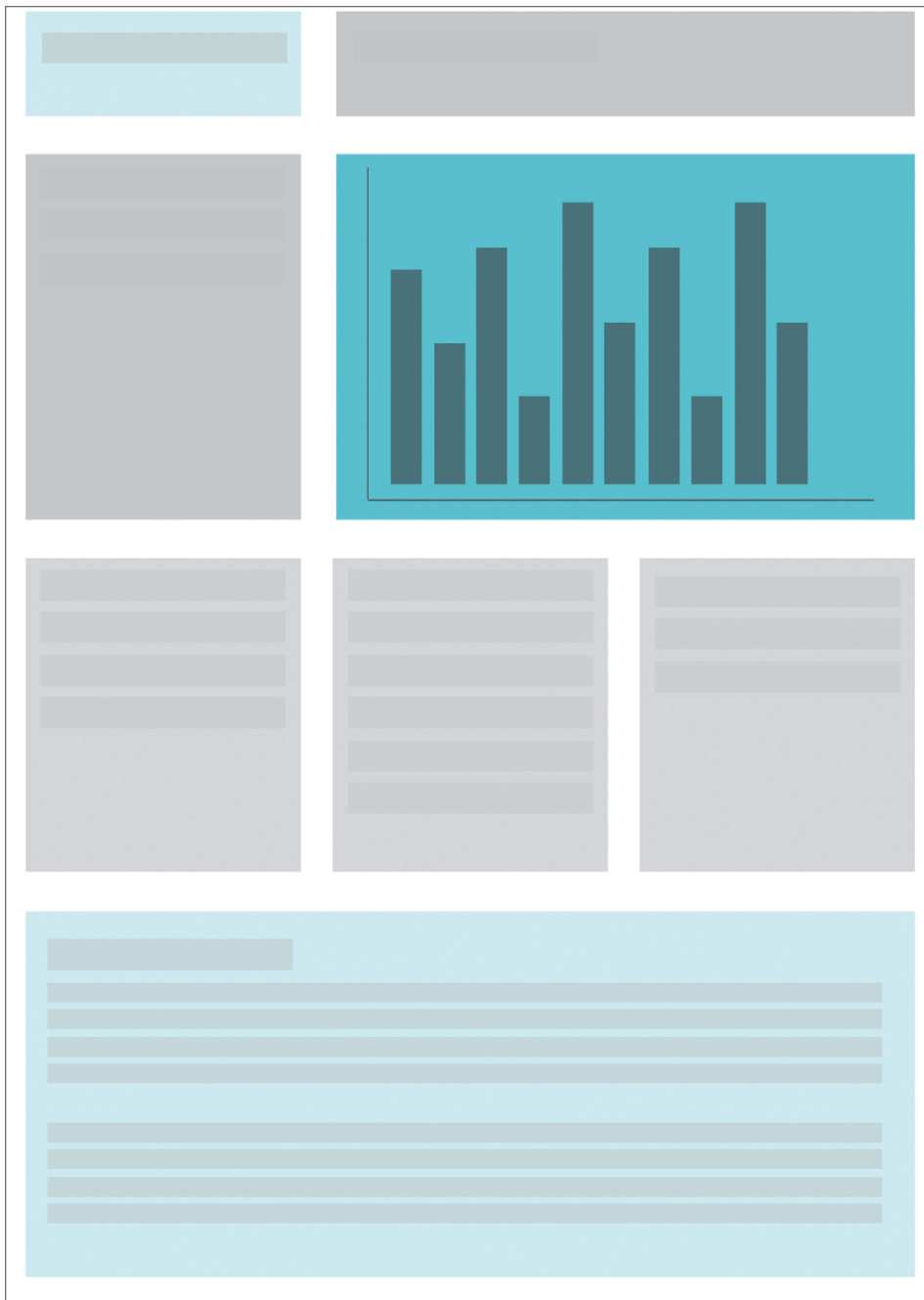


FIGURE 10-2

Ignoring the rest of the page, responding to data is a function of the elements within the visualization itself.

This concept of responding to data is what makes your visualization *fully* responsive. If you were to simply scale a visualization at different viewport sizes like an image, you would still be more responsive than a fixed-size website, but you would not be responding to all of your inputs. Tailoring the display of your data to the complexity and size of the data itself is perhaps more important than tailoring for the screen, as it is a change driven by the content itself. That is: By responding to data, you are easing the efficiency and effectiveness of understanding the data.

Putting It Together

Neither the user nor the data exists in a vacuum. There will always be unknowns on *both* the input (data) and output (client) sides of your design. Hence why you're employed.

If you combine the variables of user and data, then you need to respond to the following scenarios:

- Changing user display changes data
- Changing data granularity changes user display
- Changing user context changes both data granularity and user display

In short, any change in one variable, input or output, affects the other. Despite these two classes of unknowns, the good news is that there will always be something recognizable happening that you can use to determine how your code-driven visualization should respond.

Whether it's a window resize, a button click, or an updated API response, because you are not dealing with static content, you can update accordingly.

Using D3.js

Recall from the previous chapter that D3.js is a library of JavaScript functions for manipulating HTML documents based on data. D3 stands for data-driven

documents. Intuitively from that name, its purpose is to allow for efficient DOM manipulation based upon data inputs.

D3.js makes use of HTML and CSS standards to allow you to create markup on a page that is not contained in any sort of plugin or applet. Rather, it can be modified, styled, and interacted with in the same way that any other part of a website can be. This allows you to have much more visual and technical control over the final user-displayed result. As you'll see through the coming examples, D3's power comes from its ability to create scalable vector graphics (SVGs) in the browser.

Getting Started with D3.js

While you may want to read through the light introductions on the D3 page (d3js.org) I find that the best way to get an understanding of the library is to jump right into an example.

Importing and Understanding D3.js

Obviously, before you can use D3, you need to have it. The library can be found at <http://d3js.org/>, where you'll also find a wealth of tutorials and documentation. A comprehensive treatment of D3 would be beyond the scope of this book.

You have three ways to access D3:

Download the zip from <http://d3js.org/>.

Link *directly* to the cloud-served version of the library, which is shown in the following code, with the bolded **version** replacing the versioned release (3.5.5 as of writing):

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/d3/version/d3.min.js" charset="utf-8"></script>
```

Fork the D3 repo on github at github.com/mbostock/d3.

The first option enables you to import the latest version of the library file into your own JavaScript project as you see fit, but you'll need to go back to the website and download a new zip file whenever you want to update your version.

The second option, just as with using jQuery from the Google servers or any other library's online server, while the easiest to get going, creates a risky dependency on a third party over which you have zero control or oversight. If their end point has downtime, is moved, or is simply in a laggy mood, you have no way of fixing it.

The third option, the github repo, gives you not only the source, but also all the tests. If you download it as a zip, you'll be in just about the same situation as you would be with option one, but forking the repo using git gives you the benefits of easy version control.

It's a Library

D3.js, as I touched on in the previous chapter, is a library of JavaScript functions for manipulating HTML documents based on data. **That's all that it is.**

D3 enables you to use all the functionalities of modern browsers that you learned to harness in the first half of this book because it isn't a new technology living in a browser—it's simply JavaScript built on web technologies.

Using D3, you can *bind* data to the DOM and then apply transformations to the document based upon that data. Say, for example, you have an array of the following:

```
[0, 2, 8, 5, 1, 9]
```

You could use D3 to create an HTML table that just displays those numbers in a row. You could also use it to create an SVG bar graph of those elements. Both work using the same platform with which you've been building anything else for the web.

You don't, however, get any exciting predefined ways to represent data right out of the box. The elements available to you are just standard DOM elements. You're making HTML and styling it with CSS. Remember this sample code from the previous chapter:

```
d3.selectAll('div.change-me').style('height', function() {  
    return Math.random()*100+'px';  
});
```

Simply put, the code finds the divs with class `.change-me`, and gives them a random height. I happened to use D3 for this, but had no reason to.

With data:

```
d3.selectAll('div.change-me')
  .data([0,2,8,5,1,9])
  .style('height', function(d) { return (d * 100) + 'px'; });
```

For each element in the array, it casts to *d* in the function, and sets a height of that times (*) 100 pixels.

Selections

Because D3 is just a JavaScript library, it's technically feasible to write everything that D3 does in vanilla JavaScript.

It works, but it's *incredibly* tedious, and that's before you even back the code with any dynamic data. For example, suppose you wanted to simply change the height of all `.change-me` to 50px. Here's the vanilla JavaScript:

```
var changeUs = document.getElementsByClassName('change-me');
var changeHeight = Array.prototype.filter.call(changeUs,
function(changeMe) {
  if(changeMe.nodeName === 'DIV');
  changeMe.('style', 'height: 50px;');
}
});
```

Here it is using D3:

```
d3.selectAll('div.change-me').style('height', function() {
  return '50px';
});
```

You may have noticed that the syntax looks a lot like jQuery's. In jQuery, you would write something like this:

```
$('.div.change-me').css('height', '50px');
```

That completely makes sense. Both are using the W3C Selectors API, which is supported natively by modern browsers. Backward compatibility has been built into the selectors for both, using Sizzle in D3. Although I used a class name in the preceding example, you can select elements using any of the same selectors you would use in jQuery: tag name, containment, id, data attributes, and so on.

D3 allows you to change the nodes you've selected, setting attributes and styles. You can also add or remove event listeners, and add, remove, or sort nodes.

Moreover, you can access the DOM underlying the element directly, as a D3 selection is nothing more than an array of nodes.

Dynamic Properties

What differentiates D3 from other DOM framework libraries (such as jQuery) is that node data attributes, styles, and other properties don't need to be set as static values, but can be *functions based on data*.

Let that sink in.

You can select a bunch of nodes in the DOM, pass in some data, do whatever-the-hell-kind-of-math-you-want to it, and then change attributes of those nodes based on that data, all on the client. D3 provides a wealth of reusable functions for things such as lines, area, pie charts, graphics, and even geographic coordinates.

The first example in fact was using this:

```
d3.selectAll('div.change-me').style('height', function() {  
    return Math.random()*100+'px';  
});
```

Going one step further, suppose that for each of the elements, its height will also be a function of its place in the D3 node array, as shown in Figure 10-3:

```
d3.selectAll('div.change-me').style('height', function(d, i) {  
    return Math.random() * i * 100+'px';  
});
```

That's all crazy exciting, but what makes it useful?

Properties that are computed in these functions can be *bound* to data. Data, to D3, means an array of values passed into the function, and referenced by its first argument *d*.

By default, D3 joins nodes in the element array to their respective index in the data array. That is, the first element in the data array is passed to the first node in the selection, the second element in the data array is passed to the second node in the selection, and so on. Therefore, going back to the following data set:

```
[0, 2, 8, 5, 1, 9]
```

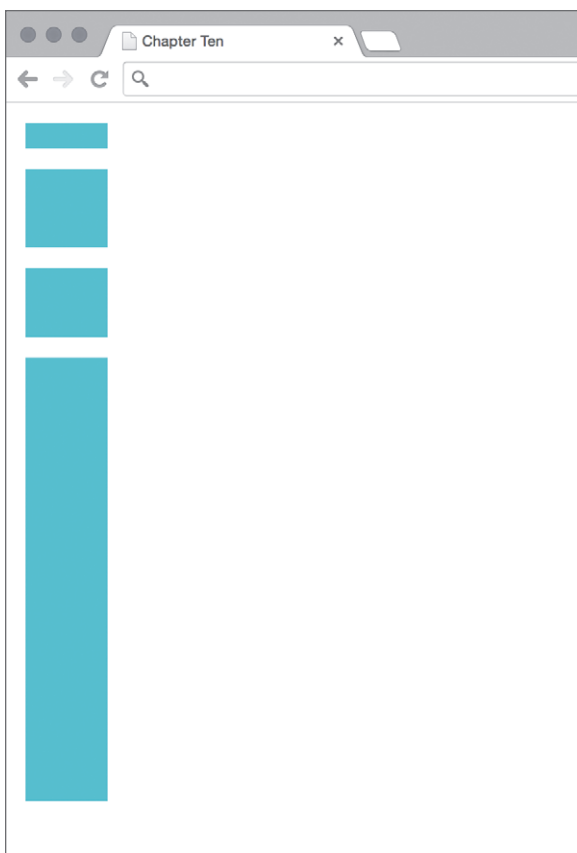


FIGURE 10-3

*Result of the D3 code using the *i* parameter*

You can see that the code for which it changed heights was actually mapping 'height: 0px;' to the first `div.change-me`, 'height: 200px;' to the second, and so on:

```
d3.selectAll('div.change-me')
  .data([0,2,8,5,1,9])
  .style('height', function(d) { return (d * 100) + 'px'; });
```

Once the data has been *bound* to these nodes, you can stop using the `data` selector, as D3 will retrieve the previously bound data. This allows recomputing (for example, when you're responding to something) without rebinding the data.

D3.js Output as SVGs

Just as you can select and modify any other type of DOM element, you have full access to SVGs. Remember that SVGs are an XML-based vector graphic format. They also support interactivity and animation. The SVG format is supported by nearly all major browsers.

That means that rather than creating and modifying the height and color of `div.change-me`, you can start to deal with things that act a lot more like the graphical elements you would expect in a chart—because they *are* the graphical elements you would expect in a chart.

Alright, but why does creating an SVG matter? To the user, the markup is irrelevant—whether it is a `div` or an SVG `rect`, they look identical.

SVG Is Great

Scalable vector graphics contain two-dimensional vectors. These can be paths, lines, shapes, or...just about any image you can make using math. Compare the images in Figure 10-4.



FIGURE 10-4

On the left: An image that works well as a raster. On the right: An image that works well as a vector

The image on the right can be represented as an SVG, the one on the left not so much. It's an image format that, in its guts, looks just like an HTML page, rather than a jpeg; and because it looks just like an HTML page, it acts just like an

HTML page, meaning you can interact with its elements using CSS and JavaScript. The biggest benefits are as follows:

- Resolution independence
- Terrific browser support
- W3C standard
- Simple to create and edit
- Compressible to tiny sizes
- Dynamically creatable with code
- Dynamically changeable with code

A plain circle can be created with the markup below, and shown in Figure 10-5:

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg" viewBox="0 0 200 200" >
  <circle fill="#32C0D3" stroke="#000000" stroke-width="5" stroke-
miterlimit="10" cx="100" cy="100" r="75"/>
</svg>
```

Neat, right?

The Ways to Build an SVG

Because an SVG is just an XML format, you have a wealth of options regarding just how you implement it within a web page. There are benefits and drawbacks to each, depending upon just what you're going to be doing with your SVG.

Assume for the following examples you have the SVG circle shown in Figure 10-5, represented by the following code, and saved as the file named `circle.tif`:

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg" viewBox="0 0 200 200" >
  <circle fill="#32C0D3" stroke="#000000" stroke-width="5" stroke-
miterlimit="10" cx="100" cy="100" r="75"/>
</svg>
```

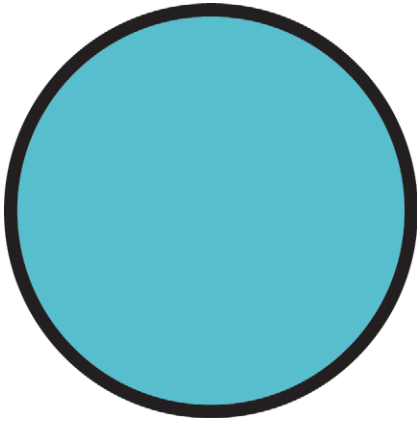


FIGURE 10-5

A circle in SVG

Img

```

```

The most obvious option is to include an SVG just as you would a PNG, JPEG, or any other image format. This method is perhaps the easiest to use, but it's also the least interactive. Apart from styles added inline (in a `style` attribute) to the SVG, it is essentially just a nicely optimized static image.

Background-image

```
.circle {  
  background-image: url("circle.tif");  
}
```

Including an SVG as a background image for an element in your CSS is also an obvious option. This has the same performance benefits as using a background image in other cases—namely, it doesn't block loading the rest of the page.

However, you are again limited to only inline and nondynamic styling, so it's best used for static content.

Iframe

```
<iframe src="circle.tif">  
  Your browser does not support iframes.  
</iframe>
```

Just as you can pull in a separate website in an `iframe`, you can do the same with an SVG. That being said...don't.

Not only would that be using the wrong tool for the job, you're introducing unnecessary complexity in your markup, and forcing an additional (also unnecessary) technical dependency upon the browser to both implement `iframes` in the first place, and to do so in the way you assume.

Object

```
<object type="image/svg+xml" data="circle.tif">
  Your browser does not support SVGs.
</object>
```

The HTML `object` tag defines an embedded object in the DOM. It's purposely generic in order to support embedding multimedia (audio, video, images, applets, PDFs, and even Flash). The unique quality of this tag is that you can also use it to embed another web page into your DOM.

In addition, because the markup of an SVG is XML, that means you're not only referencing what the outputted image looks like, you're technically embedding the SVG's markup itself.

Practically, that means you can manipulate the SVG with JavaScript, style it with external CSS, and even allow interactivity with elements.

Inline

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg" viewBox="0 0 200
200" >
  <circle fill="#32C0D3" stroke="#000000" stroke-width="5" stroke-
miterlimit="10" cx="100" cy="100" r="75"/>
</svg>
```

This is the option that you will be using most often, as D3 generates whatever elements of an SVG that you select. You'll notice that this example is the markup from the beginning of this section that drew the circle.

Rather than reference a separate file by name on your server, you can simply put the math that creates said file within your document. This has all the same benefits of using an `object` tag, but additionally doesn't obfuscate the actual element.

In a practical sense, that means you don't need to keep referencing the external file to see just what you want to modify when a user interacts with the SVG.

Which One Should You Use?

Here is the gist: For static content (such as a pre-rendered visualization), you have the option of using whatever method you are most comfortable with.

However, because you're drawing with code, your hand is forced toward inline SVG or `object` tags. Table 10-1 summarizes the limitations of each implementation method.

TABLE 10-1
Capabilities of the Four SVG Implementation Methods

	Inline	Object	Img	CSS Background-Image
CSS Manipulation	Yes	Yes	Inline	Inline
JS Manipulation	Yes	Yes	No	No
SVG Animation	Yes	Yes	Yes	Yes
Interactivity	Yes	Yes	No	No

As detailed in the preceding table, the functionality of referencing an SVG by object or by keeping all of the SVG code inline is basically identical. The key difference is that inline, you can see just what makes your SVG look the way it does.

How to Style an SVG

One of the biggest benefits of using an SVG is that you can style it using CSS.

In the code you were just looking at, the `fill` and `stroke` attributes (along with a few others) were simply added inline within the `circle` tag. Take a look at the following SVG. It's the code for the same circle but without the colors for fill and stroke.

Additionally, I've give the circle a class of `.foo`.

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg" viewBox="0 0 200 200" >
  <circle class="foo" stroke-width="5" stroke-miterlimit="10" cx="100"
cy="100" r="75"/>
</svg>
```

Now, in an external CSS file, I've added the following lines:

```
.foo {  
  fill: #32C0D3;  
  stroke: #000;  
}
```

The output, shown in Figure 10-6, is exactly what you'd expect: the same circle shown in Figure 10-5.

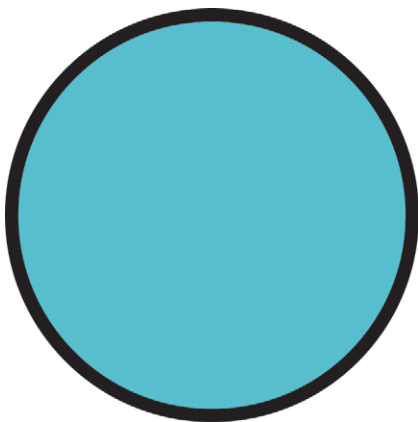


FIGURE 10-6

The same SVG circle, styled by CSS

Here's where it gets cool. Say you have a lot of these circles for some reason, and you want them to have some different styling. Rather than having to create a whole new asset, you can add a new class to the element. For example, suppose you have the following code:

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg" viewBox="0 0 200  
200" >  
  <circle class="bar" stroke-width="5" stroke-miterlimit="10" cx="100"  
cy="100" r="75"/>  
  
  <circle class="bar" stroke-width="5" stroke-miterlimit="10" cx="100"  
cy="100" r="75"/>  
</svg>
```

Here is the CSS:

```
.foo {  
  fill: #32C0D3;  
  stroke: #000;  
}  
.bar {  
  fill: #00AA55;  
  stroke: #999;  
}
```

The output, shown in Figure 10-7, demonstrates that by using identical markup to set up the SVG, but adding a class, you can radically change its styling. How this works in a practical sense is the same as how CSS applies to any other part of the DOM. Remember the priority order of styling, as shown in Table 10-2.

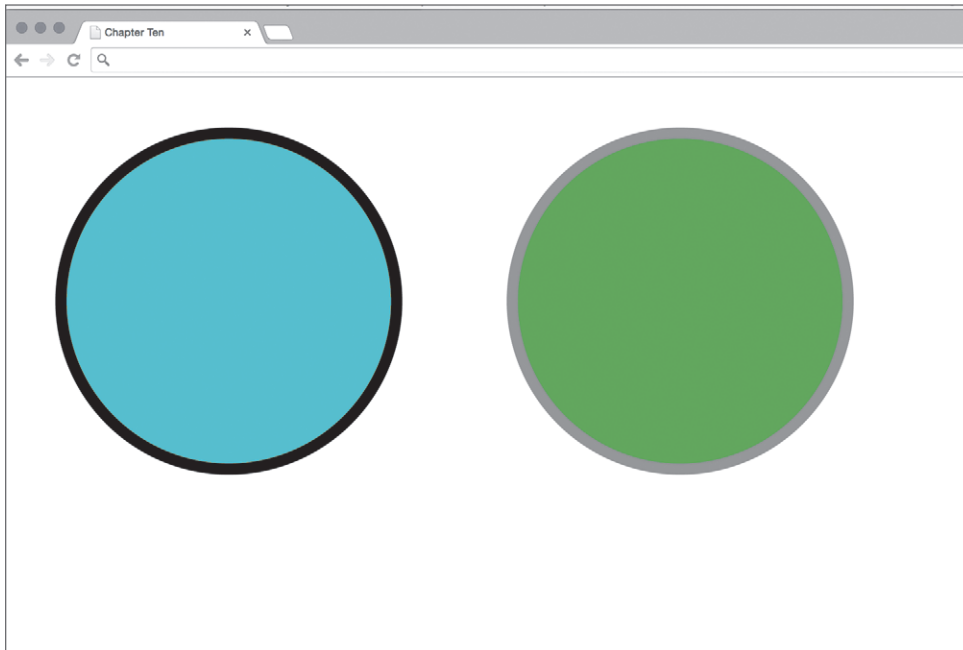


FIGURE 10-7

Two SVGs, both styled by CSS. Although their markup is the same, they have separate classes that result in different styling.

TABLE 10-2

CSS Source Type Priority Levels

Priority Level	Style Source Type	Style Type Description	Example
0	Importance	Any rule either in a stylesheet or inline HTML with the <code>!important</code> modifier	<pre>h1 { font-size: 24px; !important; }</pre>
1	Inline	Rules defined in an HTML element's <code>style</code> attribute	<pre><h1 style="color: red;">...</h1></pre>
2	Media type	Rules applied to specific media types as defined in the CSS	<pre>@media screen { h1 { color: red; } }</pre>
3	User defined	A function of the browser, allowing users to define their own CSS overrides	N/A
4	Selector specificity	A specific contextual selector that overwrites the generic definition	<pre>#header h1 { color: blue; }</pre>
5	Rule order	Rules defined last in the CSS are given higher priority than those defined first.	<pre>h1 { color: red; } h1 { color: blue; /* This is higher priority */ }</pre>
6	DOM parent inheritance	A property inherited by an element that has been defined on the element's parent	<pre>#heading { font-size: 24px; } h1 { color: red; /* h1 inside #heading will also inherit font-size: 24px; */ }</pre>
7	Generic CSS property	CSS rule or inline style that overwrites a browser default	<pre>h1 { color: red; }</pre>
8	Browser default	The default browser value as determined by W3C initial value specifications	N/A

How does all of this apply to responsive design?

You spent all of Chapter 4 learning how to master the media query to target your styles to different device functionalities. That doesn't change here. Assume the following code again:

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg" viewBox="0 0 200
200" >
  <circle class="bar" stroke-width="5" stroke-miterlimit="10" cx="100"
cy="100" r="75"/>
</svg>
```

Now apply some CSS with a media query, as here:

```
.foo {
  fill: #32C0D3;
  stroke: #000;
}
@media screen and (min-width: 600px) {
  .foo {
    fill: #000;
    stroke: #FFA905;
  }
}
```

As shown in Figure 10-8, you see that the circle has the normal blue styling; but on a screen size > 600px, the additional styling changes its appearance.

Obviously this is a pithy example, but assume you have a big data set being represented as a bar chart. If on a small screen you're *grouping* down to only three wide bars with labels shown, and on a massive television you have 300 small bars with *layered* information on interaction, you can skip any reliance on JavaScript and simply build this into your media-queried CSS.

Creating SVGs with D3.js

At this point you should understand the benefits of SVG, and you've seen how incredibly simple it is to add, remove, and modify elements on the DOM using D3's built-in functions.

It's time to put that knowledge to use.

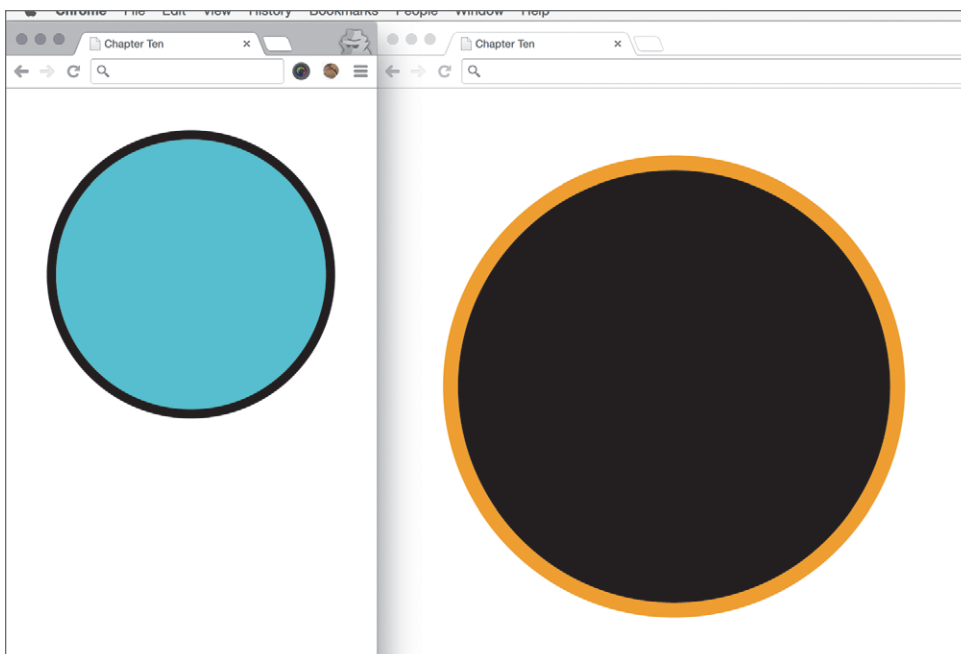


FIGURE 10-8

An SVG circle that changes color when the screen size exceeds 600px

Outputting Four SVG Squares

Suppose your DOM includes an SVG with four squares (`rect` tags with equal width and height), built with the following code (and shown in Figure 10-9):

Code

```
<svg width="550" height="100">
  <rect x="100" y="25" width="50" height="50" />
  <rect x="200" y="25" width="50" height="50" />
  <rect x="300" y="25" width="50" height="50" />
  <rect x="400" y="25" width="50" height="50" />
</svg>
```

There's nothing exciting about this so far—it's just four squares of equal size in a line. For the time being, they're each 50px by 50px, vertically centered within the SVG, and 50px apart from one another.

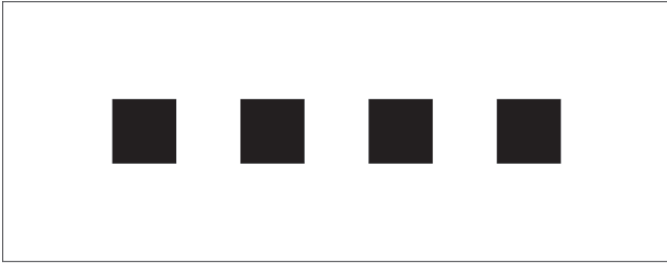


FIGURE 10-9

The original four SVG squares

Selections

The `d3.selectAll` method accepts a selector, which can be a class, tag name, containment, id, data attributes, and so on. In this case, you can use `'rect'` to select all four of the squares, as shown here:

```
var squares = d3.selectAll('rect');
```

The variable `squares` now references an array, including all four of the `rect` nodes. With that selection, you can start to change the nodes, just as you did to `divs` earlier in this chapter. For example, you can change the color of all of them:

```
squares.style('fill', '#32C0D3');
```

Applying that rule sets the same style of a blue fill for all elements within the array. The markup is updated as follows, and visually the SVG is changed to what is shown in Figure 10-10:

```
<svg width="550" height="100">
  <rect x="100" y="25" width="50" height="50" style="fill:#32C0D3;" />
  <rect x="200" y="25" width="50" height="50" style="fill:#32C0D3;" />
  <rect x="300" y="25" width="50" height="50" style="fill:#32C0D3;" />
  <rect x="400" y="25" width="50" height="50" style="fill:#32C0D3;" />
</svg>
```

Suppose that instead you want to traverse through the array and modify each element separately. You can do so by using an anonymous function within the selection. Anonymous functions are used in D3 to compute attribute values for scale and shape when related to bound data. More on that in a moment.

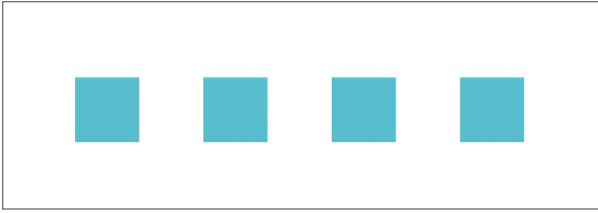


FIGURE 10-10

The four squares are now blue.

For this example, just assume you want to change the squares into rectangles of varying size with a maximum width of 50px. You can do so with the following code:

```
squares.attr('width', function() { return Math.random()*50; });
```

Notice the use of the `Math.random()` function to create a random modifier for the 50 pixel maximum. Upon running this code, the markup is updated as follows, and the SVG looks like Figure 10-11:

```
<svg width="550" height="100">
  <rect x="100" y="25" width="33.369975339155644" height="50"></rect>
  <rect x="200" y="25" width="13.0528430105187" height="50"></rect>
  <rect x="300" y="25" width="43.3788719587028" height="50"></rect>
  <rect x="400" y="25" width="17.016702063847333" height="50"></rect>
</svg>
```

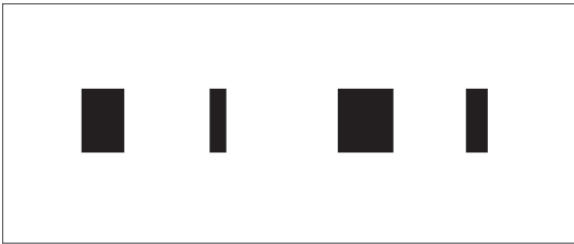


FIGURE 10-11

The four squares are now rectangles with random widths, based on the D3 function applied.

Data

Using only random numbers could be handled just as easily in jQuery, or even vanilla JavaScript. The key benefit of D3 over another DOM manipulation library

is that it enables you to *bind* data to the DOM and then apply transformations to the document based upon that data.

Therefore, still using the same four squares, say you now have the following code:

```
var squares = d3.selectAll('rect');  
squares.data([7,13,19,25]);
```

Nothing has changed on the DOM yet, but something important has happened behind the scenes. In the first line, you have once again selected the DOM elements. In the second, you have taken in an array of four values that essentially mirrors the selection on the data side.

That means the first number (7 in this case) is now *bound* to the first square in the selection.

After data is bound like this, it's accessible to the selection's functions. As a best practice, the datum bound to each selected element is named *d* when passed into the anonymous function.

Assuming you've run the two preceding lines, you can now change the rectangles' widths using *d*, as shown here:

```
squares.attr('width', function(d) { return d*2; });
```

The updated markup is as follows, and the SVG is shown in Figure 10-12:

```
<svg width="550" height="100">  
  <rect x="100" y="25" width="14" height="50"></rect>  
  <rect x="200" y="25" width="26" height="50"></rect>  
  <rect x="300" y="25" width="38" height="50"></rect>  
  <rect x="400" y="25" width="50" height="50"></rect>  
</svg>
```



FIGURE 10-12

The rectangles now have widths backed by their bound data. Although primitive, this is a data visualization.

There's one more argument that is typically useful when modifying selections: the index of an element within the array. By convention, it's named `i`.



NOTE

Hey, superstar. You just used D3 to visualize data as an SVG.

In the same example, say you'd like to change the height of the rectangle based upon its position in the array. You can do that with the following code:

```
squares.attr('height', function(d,i) { return (i+1)*10; });
```

The markup ends up as follows, and the visualization is shown in Figure 10-13:

```
<svg width="550" height="100">
  <rect x="100" y="25" width="14" height="10"></rect>
  <rect x="200" y="25" width="26" height="20"></rect>
  <rect x="300" y="25" width="38" height="30"></rect>
  <rect x="400" y="25" width="50" height="40"></rect>
</svg>
```

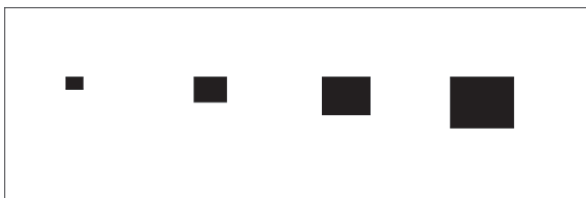


FIGURE 10-13

The rectangles now change size based upon both their bound data and their index in the selection array.

Everything else starts to spawn from this basic set of functionality. Given elements, an array of data, and the two traversal variables of `d` and `i`, you can select any element and modify its attributes based upon the data that you've bound to it.

At a certain point, however, the data that was initially bound might not be the data that you want to show. In the previous chapter you used `enter` and `exit` to handle cases in which you wanted to add or remove data.

Now you're going to dig deeper into how these two selections work.

Enter

Assume your four squares are back to being just squares; but instead of your data being a set of `[7, 13, 19, 25]`, it was now `[7, 13, 19, 25, 11]`. The problem is that you have one more datum than you have selected elements. You now need to find a way to append an additional node to the DOM in order to represent the fifth datum.

You can certainly just manually update the DOM, provided you know that this change is coming, and doesn't happen on the fly. In the real world, however, you won't know this change is coming, and it *will* happen on the fly. You solve for this using `enter`.

When you join that data to the selected elements, each datum can end up in one of two places:

- `update`: The joined nodes of a DOM element and its data
- `enter`: The overflow data that doesn't have an element to bind to

Therefore, if you ran the new array of five elements through the previous code, the output would look the same, but behind the scenes that eleven would end up in an overflow bucket of enterable data.

This is good news. It means that the additional piece of data hasn't just floated off into the ether—it's available for use. By appending the `enter` selection to the DOM, you can create a new rectangle from the missing data.

In terms of creating rational markup, you'll now need to start with a reference to the `rects`' parent element, because you'll want to inject a new element at the same level as the others. This is simply the wrapping SVG. The following example shows what the beginning of your D3 function needs to look like to handle this new selection:

```
var svg = d3.select('svg');  
  
var squares = svg.selectAll('rect')  
    .data([7, 13, 19, 25, 11]);  
var enterSquare = squares.enter().append('rect');
```

When you `enter` an element, it already is bound to the data, so you don't need to re-data anything to reference `d` or `i`. As such, setting the attributes for the new element is the same as what you did for `squares` earlier:

```
enterSquare.attr('x', function(d, i) { return (i+1)*100; });
```

```
enterSquare.attr('y', function(d, i) { return 25; });
enterSquare.attr('width', function(d) { return d*2; });
enterSquare.attr('height', function(d,i) { return (i+1)*10; });
```

Running all of this, you get the following markup, and the SVG shown in Figure 10-14:

```
<svg width="550" height="100">
  <rect x="100" y="25" width="50" height="50"></rect>
  <rect x="200" y="25" width="50" height="50"></rect>
  <rect x="300" y="25" width="50" height="50"></rect>
  <rect x="400" y="25" width="50" height="50"></rect>
  <rect x="500" y="25" width="22" height="50"></rect>
</svg>
```

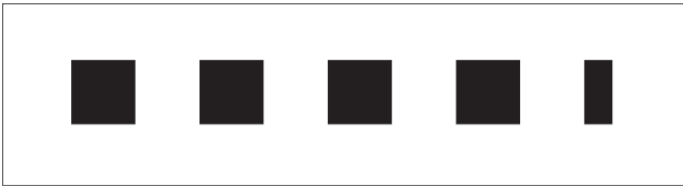


FIGURE 10-14

This isn't right. The original squares didn't change.

That's...not right. The problem is that you started with four `rect` elements on the page, but in the new code you're modifying only the attributes of the `enter` selection, cast to `enterSquare`. The entire code needs to handle both selections, as shown here:

```
var svg = d3.select('svg');

var squares = svg.selectAll('rect')
  .data([7,13,19,25,11]);
squares.attr('width', function(d) { return d*2; });
squares.attr('height', function(d,i) { return (i+1)*10; });

var enterSquare = squares.enter().append('rect');
enterSquare.attr('x', function(d, i) { return (i+1)*100; });
enterSquare.attr('y', function(d, i) { return 25; });
enterSquare.attr('width', function(d) { return d*2; });
enterSquare.attr('height', function(d,i) { return (i+1)*10; });
```

This finally gives you the following markup, and Figure 10-15, exactly what you wanted to visualize:

```
<svg width="550" height="100">
  <rect x="100" y="25" width="14" height="10"></rect>
  <rect x="200" y="25" width="26" height="20"></rect>
  <rect x="300" y="25" width="38" height="30"></rect>
  <rect x="400" y="25" width="50" height="40"></rect>
  <rect x="500" y="25" width="22" height="50"></rect>
</svg>
```



FIGURE 10-15

That's more like it. By using both data and enter, the rectangles have been properly drawn.

If you are thinking that this code is extremely wasteful, you're right. There's a ton of overlap in functionality, when all you're really trying to do is make all the elements match their data. In the case where your entire selection ends up in the `enter` bucket, you can chain together all these functions as shown in the following example:

```
var svg = d3.select('svg');
svg.selectAll('rect')
  .data([7,13,19,25,11])
  .enter().append('rect')
  .attr('x', function(d, i) { return (i+1)*100; })
  .attr('y', function(d, i) { return 25; })
  .attr('width', function(d) { return d*2; })
  .attr('height', function(d,i) { return (i+1)*10; });
```

That said, however, you will have the same issue of four of the five squares still being squares if they are in the `update` bucket, rather than `enter`.



WHY BOTHER WITH BOTH?

This was a unique case because you were working with previously created elements. In most cases, you'll be building every element within D3, so you won't have this apparent overlap between data and enter.

Exit

The opposite problem of having more data than elements is obviously having more elements than data to represent. This is the easier situation to handle. Your markup is again the four squares, and your data is now `[7, 13, 19]`. The fourth element should no longer be present in the DOM. Skipping the `enter` selector for a moment, your code to handle this looks like the following:

```
var svg = d3.select('svg');

var squares = svg.selectAll('rect')
    .data([7, 13]);

squares.attr('width', function(d) { return d*2; });
squares.attr('height', function(d,i) { return (i+1)*10; });

squares.exit().remove();
```

That last line takes care of removing elements to which data is no longer bound. The markup is simply three elements now:

```
<svg width="550" height="100">
  <rect x="100" y="25" width="14" height="10"></rect>
  <rect x="200" y="25" width="26" height="20"></rect>
  <rect x="300" y="25" width="38" height="30"></rect>
</svg>
```

The SVG is shown in Figure 10-16.

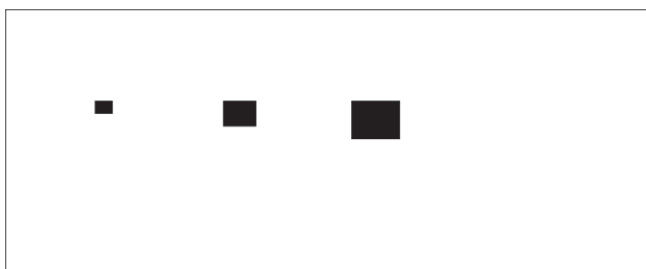


FIGURE 10-16

After exit, only three elements remain.

Scale

Scales in D3 are functions that map from an input domain to an output range. In non-robot language, your scale is what translates your data's value to its representative size on the screen. There are two types of D3 scales: quantitative and ordinal.

- **Quantitative** scales have a *continuous* domain, the simplest example being numbers.
- **Ordinal** scales have a *discrete* domain, such as a list of state names.

The actual scale function of D3 is optional; it is essentially a nice wrapper around the math you would have to do to create your own mapping of data to visual representation. For the purposes of this chapter, you'll be mapping quantitative scales, as they have a much clearer 1:1 relationship between input (numbers) and output (numbers).

Domain

Your domain is the set of possible values for your input data. Suppose you start with the initial data set shown here:

```
var dataset = [ 0, 1000, 2000, 3000, 4000 ];
```

The domain of this set is a minimum of 0 to a maximum of 4000. Up until this point, that data has been what you have used as the *d* in your functions to actually display on the screen.

Therefore, for example, looking at the previous squares example, your first element's width would be 2000px, and your last element's width 4000px. That's not going to work very well on a small screen, or even a large screen, for that matter. Moreover, suppose the data set grows, and each additional datum is 1000 more than the previous. Your set will become completely undisplayable very quickly.

This is the purpose of a scale. Because this data is not meant to represent pixels on a screen, you don't need to force it to do that.

Range

Range is the set of possible output values of your visualization. Using D3 on the web, that means the pixel values that map your data to the screen. Unfortunately,

you can't just automatically use percentages for a D3-drawn chart. Pixels have to stay involved.

As great as it would be to build an entirely fluid chart, it's just not how the library works. Start with a simple stacked bar chart that isn't responsive for now.

Suppose your container is a fixed 400px wide, with padding on either side of 8px. Your range for this set is 8 to 392. Table 10-3 shows the data set's values mapped to their ranges.

TABLE 10-3

Domain Data vs. Computed Pixel Range

Data	Pixels
0	0
1000	98
2000	196
3000	294
4000	392

In Chapter 9, you actually used this code already:

```
...
var x = d3.time.scale()
    .range([8, 304])
...
var y = d3.scale.linear()
    .range([144, 8])
...
var svg = d3.select('#viz').append('svg:svg')
    .attr('width', 320)
    .attr('height', 160)
...
```

There, your `range_x` was 8–304px, your `range_y` was 8–144px, and your domain was as follows:

```
d3.extent(data)
```

This function simply acts as a D3.js wrapper to output min and max values of a data set.

What about handling the output responsively, though? When you were first learning how to create fluid layouts, recall the equation that defined a percentage of the screen:

$$\text{target} \div \text{context} = \text{result}$$

If the result is the fluid percentage-based size of your SVG (or its container), then at every pixel value of your context (the screen size), you do still have a pixel-based target. That is:

$$\text{target} = \text{result} \times \text{context}$$

At least at one size, you can determine exactly what your range should be to fix your container. Keep this in the back of your mind for a few minutes.

Transition

A transition is simply a way to animate and smooth the shifting of data. On enter, exit, and scale shifts, the default behavior is to simply instantaneously redraw. Transitions delay and animate these changes over fixed (and modifiable) durations.

Directly from the D3.js documentation:

A transition is a special type of selection where the operators apply smoothly over time rather than instantaneously. You derive a transition from a selection using the transition operator. While transitions generally support the same operators as selections (such as attr and style), not all operators are supported; for example, you must append elements before a transition starts. A remove operator is provided for convenient removal of elements when the transition ends.

Recall from the previous chapter that when you needed to change the scale as elements were added and removed, pushing the limits of your domain, you used the following function:

```
var redraw = function() {  
  x.domain(d3.extent(data, computeDate));  
  y.domain(d3.extent(data, computeVal));  
  
  var items = svg.selectAll('circle').data(data);  
  
  items  
    .attr('cx', function(d) { return x(computeDate(d)) })
```

```

    .attr('cy', function(d) { return y(computeVal(d)) });

    items.enter()
      .append('svg:circle')
      .attr('r', 4)
      .attr('cx', function(d) { return x(computeDate(d)) })
      .attr('cy', function(d) { return y(computeVal(d)) });

    items.exit()
      .remove();
  };

```

To add a transition, only the following change (in bold) needs to be made:

```

var redraw = function() {
  x.domain(d3.extent(data, computeDate));
  y.domain(d3.extent(data, computeVal));

  var items = svg.selectAll('circle').data(data);

  items
    .transition()
    .attr('cx', function(d) { return x(computeDate(d)) })
    .attr('cy', function(d) { return y(computeVal(d)) });

  items.enter()
    .append('svg:circle')
    .attr('r', 4)
    .attr('cx', function(d) { return x(computeDate(d)) })
    .attr('cy', function(d) { return y(computeVal(d)) });

  items.exit()
    .remove();
};

```

Beyond what you have learned here, D3.js offers a lot more functionality. It isn't covered here because none of it relates to making visualizations responsive. However, the Resources section at the end of the book has a wealth of tutorials and articles that can set you on the right path.



SLIMMING DOWN

Animation can't be shown in a book, so head to this chapter's section on responsivedatavisualization.com and check out the example.

Data and the DOM

D3 is the translation layer between your structured data and its representation in the DOM. It requires an array of values that can be somehow sorted and individually parsed, but apart from that it does not force many rules upon you.

In Chapter 8, you learned how to create a clean, powerful API. Here is where the work you have front-loaded there becomes immensely useful. Your visualizations are going to be powered by data coming from your API, and parsed via JavaScript. Before combining D3 and your API, we'll dig into what we mean when we refer to data in JavaScript.

Data in JavaScript

You're still just using a JavaScript library, so you are forced to use JavaScript. And that's good, because it's a wonderful, powerful, Wild West of a language.

In Chapter 5, you learned that JavaScript is a loosely typed language. Practically, that means you can set a variable to be any type of object or literal, and then change it to another without a problem:

```
var myString = 'Hello world';
myString = 1;
myString = [1,2,3];
myString = {
  'name': 'Hello world',
  'val': 1
};
```

Everything in the preceding code will work normally. The value of `myString` at the end is now an object with fields `'name'` and `'val'`.

Your data, if you have built an API based upon the standards detailed in Chapter 8, should be reachable as a JSON (JavaScript object notation) file. This notation is essentially a large object (like what `myString` ended up as above) that can be parsed into more digestible and displayable information, like strings or numbers.

Although there isn't really a concept of strongly-enforced data types, there are still reasonable data types that allow for operations on variables. Without data types,

there would be no way to compare, combine, or parse different pieces of data. These data types are detailed in the proceeding sections.

Booleans

True or false. This is conceptually the simplest data type. Booleans are used in conditional testing and can only hold a value of true or false:

```
var x = true;
var y = false;
```

Numbers

JavaScript's numbers can hold either an int (no decimal) or a float (decimals). Numbers can also be written in scientific notation:

```
var x = 1;
var y = 1.000;
var z = 1e11;
```

Strings

Strings are a series of characters, and are written with either single or double quotes:

```
var x = 'Hello world';
var y = "Hello world";
```

Arrays

Arrays are written using square brackets, and items are separated by commas. Remember that D3 takes in an array of items for its `data` selector:

```
var arr = [1, 55, 19];
```

Apart from simple data types, arrays can hold objects, or even additional arrays:

```
var obj_arr = [
  {
    'name': 'Object 1',
    'val': 123,
  },
  {
    'name': 'Object 2',
    'val': 456,
```

(continues)

(continued)

```
    },  
    {  
      'name': 'Object 3',  
      'val': 789,  
    },  
  ],  
];  
var arr_arr = [[1,2,3],[4,5,6],["Hello world","Goodbye world"]];
```

Objects

Objects in JavaScript can be thought of as variable wrappers around more variables. An object can look something like this:

```
var obj = {  
  'name': 'Object 1',  
  'val': 123,  
  'id' 55,  
  'related': [11,2,40]  
}
```

Objects can also hold other objects (and arrays of other objects). The attributes of one object can be referenced either by dot or square bracket notation. In the preceding object, you could retrieve the value of `name` using either of the following:

```
obj.name;
```

or

```
obj['name'];
```

An important type of object that is used often is the date object, which comes with a whole pile of useful functions. Date objects have a constructor of `new Date()`, which defaults to the exact date and time when it is called, but it can accept parameters to change year, month, day, hour, second, and millisecond.

JSON

Finally, JSON (JavaScript object notation) is actually a file type that is formatted as a JavaScript object. As such, the elements within it are simply objects, which can be accessed and traversed in the same way. You have already dealt with JSON a lot in this book. If you need a refresher, take a look at the train tracker example in Chapter 5.

Structuring Your Data

Unless you're writing examples for a book on responsive data visualization, your data is rarely going to be sitting idle and unchanging in a variable within your JavaScript. You will be reaching out to an API and retrieving JSON.

In Chapter 8, you learned that a good set of data should be all of the following:

- Searchable
- Sortable
- Machine-processable

That is, your data should always answer the following three questions with a resounding “Yes!”:

- Can someone search through the data and find something for which they are purposefully looking?
- Can related fields across items be compared and evaluated to a greater or lesser result, enabling uniform items to be ranked by an individual field, or a join of multiple fields?
- Without human intuition, can the data be scrubbed and analyzed?

Once your data can satisfy these conditions, you can build your API to produce the exact output you're going to feed into your visualization, without any bloat or unnecessary bandwidth consumption.

A Good API

Your API should have the following characteristics in order to be best consumed by D3.

Granular Access

You should be able to get to the most granular level of useful data within your set.



SLIMMING DOWN

The following sections represent a subset of the list outlined in Chapter 8 describing the attributes of a good API. It's not that the other elements aren't important, they're just less important to visualizing with D3.

Just as important as what is returned, granular access to your data via your API means that anything outside of the requested outputs should *not* be returned, in order to reduce noise and bandwidth.

Deep Filters

If the granular item is the needle you're looking for in your haystack of data, filters are the magnets you use to find it. Searching through the full text of any data set would be ludicrous, and take considerable time, so an API should be able to take in pre-populated or free-form text filters to narrow your searchable set.

Value Typing

You don't want to be stuck doing expensive comparisons on the client to calculate and identify data, so your returned array of objects, and their attributes, should be of an obvious and useful type.

RESTful

Because you're doing this in JavaScript, the easiest and most practical way to reach your data is at a RESTful end point. This means that the granular access and deep filtering built into your API should be constructible as a RESTful string.

You saw this most clearly in Chapter 5's train tracker API. A URL to the list of stops on the blue line was as follows:

```
"/blue"
```

And this was a stop with ID 40890 in the collection of stops on the blue train line:

```
"/blue/40890"
```

Remember that, in essence, a RESTful API is just a URL to your data.

Standardized Format

Use JSON. It's the most easily consumed data source by D3. It's also dead-simple to construct.

High Availability and Performance

This is especially important as your data set becomes more complex, and performance on the client begins to waver. You cannot expect users to first wait for your code to fetch data from a slow server, and then also be slow at processing and displaying it.

Here are a few tips to ensure high availability and performance:

- Don't allow for accidentally complicated queries by URL.
- Don't put resource-heavy authentication in the URL.
- Cache results on the server.
- Cache repeated queries on the client.

How D3 Reads Data

D3 takes in data as an array. That's it. That's the only option you have. However, that shouldn't keep you from throwing big complicated data sets at it anyway. A JavaScript array can hold objects with as many attributes as you want. In practice, you don't need to limit yourself to something like this:

```
var dataset_a = [10,20,55,101];
```

You can instead do something along the lines of the following:

```
var dataset_b = [  
  {  
    'id':341,  
    'name':'Aldo',  
    'val':25  
  },  
  {  
    'id':102,  
    'name':'Amber',  
    'val':105  
  },  
  {  
    'id':499,  
    'name':'Bertram',  
    'val':6  
  },  
  {
```

(continues)

(continued)

```
'id': 790,  
'name': 'Bill',  
'val': 99.123  
}  
]
```

This creates the need for the D3 *key* function. By default, D3 simply reads elements by index, meaning that in `dataset_a`, datum 1 is 10, datum 2 is 20, and so on; and in `dataset_b`, datum 1 is the object with `id` 341, datum 2 is the object with `id` 102, and so on. That happens when you call `data` without an additional function, like this:

```
var people = svg.selectAll('rect')  
  .data(dataset_b);
```

A key function, passed to the data selector, enables you to specify what should *actually* identify a unique piece of data. A suitable key function for `dataset_b` would be to take each object's `id`. Within a visualization's backing code, that would look like this:

```
var people = svg.selectAll('rect')  
  .data(dataset_b, function(d) { return d['id']; });
```

The biggest benefit of a key function is that it solves for duplicate entries in your data. As a fringe benefit, this data consistency allows for much more meaningful transitions and interaction. Adding or removing, sorting, and changing layering between data will transition the correct elements because there is a piece of the data more meaningful than its index to connect two views.

Building Responsive Data Visualization for the Web

Get it? Like the book's name.

Now that you have identified all the unknowns for which you have to solve (or at least guard against), and you have learned the tools with which to do so...do so.

At the beginning of the chapter, you learned that the two unknowns you need to respond to are “client” and “data” unknowns. The first reflects changes on the output end, the second changes on the input end.

Client Responsiveness

This is what you think of when you think of responsive design—making things work on all screen types. And it’s an incredibly important half of the equation. You are creating these visualizations *for someone*, and the whole point of all this effort is to enable that someone to look at your visualization on any device.

Using Your CSS

Before getting into the more data-visualization-specific JavaScript, keep in mind that you are still using markup and can still apply classes to that markup. Although there isn’t much you can do to totally resize your visualization in CSS, you can actually *decrease* the amount of work you need to do on the visualization end by more methodically using CSS to cause the rest of the surrounding page to respond and remain fluid.

Look back to Figures 10-1 and 10-2 at the beginning of this chapter. The CSS is an obvious choice for reflowing and resizing the visualization’s container (Figure 10-1), and how it interacts with the page as a whole. Responsive CSS can also help with *some* of the visualization-internal changes (Figure 10-2) you may want to make, such as font size, padding between elements, and hiding or showing layered elements.

Window Resize

CSS, however, is only going to get you so far when you’re building things dynamically with JavaScript. Assume some arbitrary large data set (available at responsivedatavisualization.com/chapter-ten) that can be fed into a bit of D3 code to create the stacked bar chart shown in Figure 10-17.

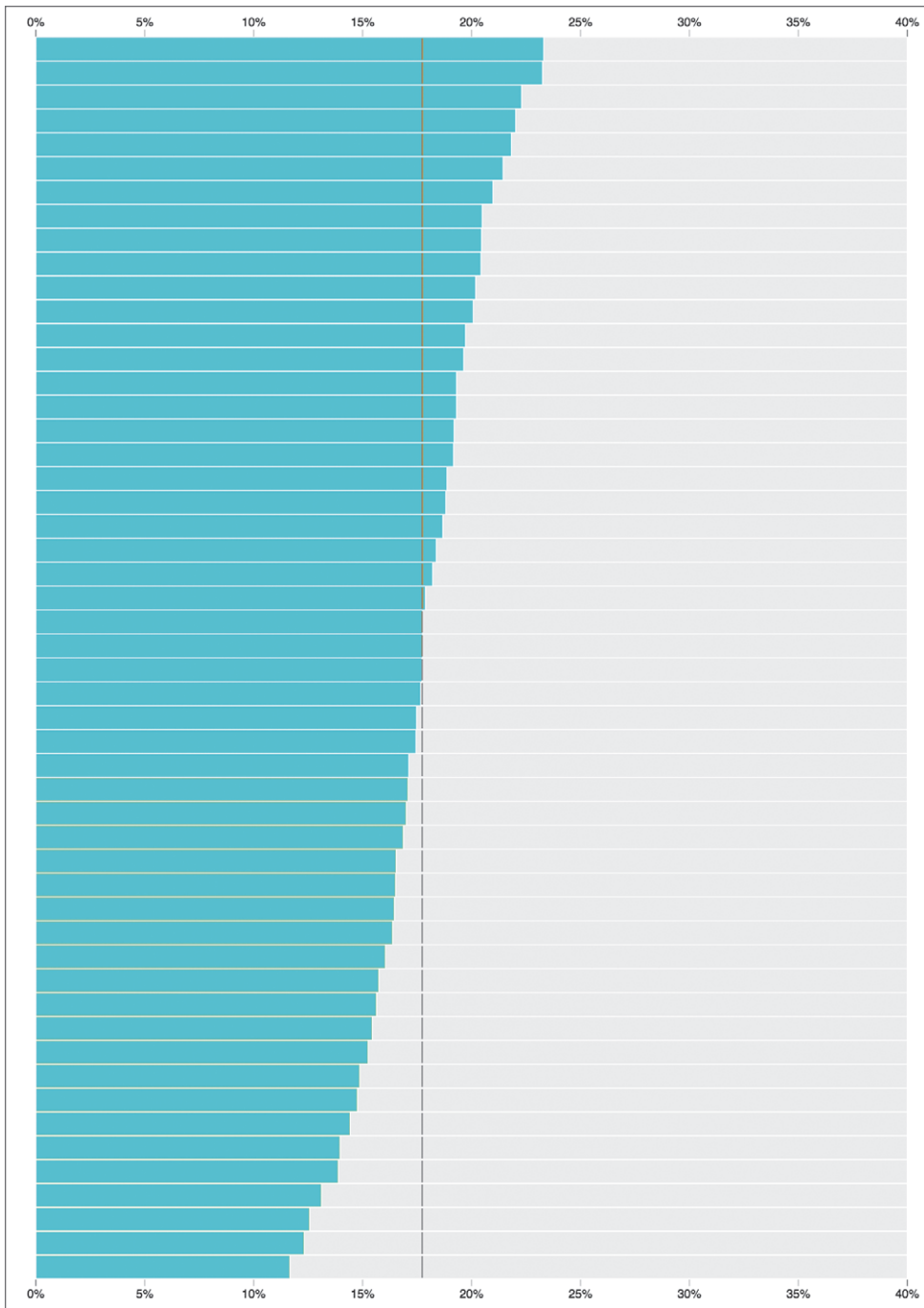


FIGURE 10-17

A nonresponsive bar chart

Instead of setting the range as some number, I'm going to actually take it from the container's width. Suppose the DOM looks like this:

```
<header>...</header>
<main>
  <section class="wrapper">
    <div id="viz"></div>
  </section>
  <aside>...</aside>
</main>
<footer>...</footer>
```

In addition, it has some basic fluid styling:

```
header,
footer {
  width: 100%;
}
main {
  @include clearfix();
  max-width: 1024px;
  width: 100%;
}
.wrapper {
  float: left;
  width: 60%;
}
aside {
  float: left;
  width: 40%;
}
```

The code in D3 to figure out the chart's size could then, rather than simply filling a 400px by 100px, as in the previous examples, actually query the size of the container:

```
var margin = 8;
var width = d3.select('#viz').style('width');
var width = width - (2*margin);
var barHeight = 24;
var percent = d3.format('%');

var x = d3.scale.linear()
  .range([0, width])
```

(continues)

(continued)

```
.domain([0, 1]);  
  
var y = d3.scale.ordinal();
```

Your visualization, at least at page load, is now resolution-agnostic. Both the chart's *size* and *scale* are dependent upon the width of `#viz`.

Resizing All the Time

The next step should be intuitive. If you can resize on page load, you can do it any-time the page size changes:

```
function redraw() {  
  width = d3.select('#viz').style('width');  
  width = width - (2*margin);  
  
  x.range([0, width]);  
  
  d3.select(chart.node().parentNode)  
    .style('height', (y.rangeExtent()[1] + 2*margin) + 'px')  
    .style('width', (width + 2*margin) + 'px');  
  
  chart.selectAll('rect.percent')  
    .attr('width', function(d) { return x(d.percent); });  
}
```

One major drawback of this technique is that it can become expensive if you are redrawing every time the page fires a resize event. The obvious solution is to only redraw when you are actually changing page content.

Matching Your Breakpoints

Just as you don't make radical page reflow changes at every single pixel change, nor do you need to handle your visualization this way. In addition, just as you found breakpoints at which other content needed to be reflowed, so too can you set this for your visualization.

The simple solution is to take your major breakpoints from your CSS, as they are presumably causing the container for your visualization to change, and only redraw then.

The more complex (but certainly worthwhile) solution is to do the same thing you did when designing a responsive page. Let the content show you what's wrong. Build for your smallest supported screen; then, as you find a spot at which it makes

sense to redraw, create a new breakpoint there. How you handle drawing at these breakpoints is a concept called *drawing modes*, which you'll now add to your redraw function.

It works like this: In CSS your styles apply once a media query is met, and then they don't need to be re-applied because that media query is still met. However, in JavaScript, if you want to make a change when the page is, for example, > 600px wide, and you change the page from 580px to 601px, and then change 601px to 625px, the redraw will fire both times.

Therefore, you'll set a variable that describes the view mode in which your user is viewing the content. The following skips the guts of the redraw just outlined:

```
var mode;
handleResize();
d3.select(window).on('resize', handleResize);

function handleResize() {
  var windowWidth = window.innerWidth;
  if(windowWidth >= 1024 && mode !== 'lg') {
    mode = 'lg';
    redraw(mode);
  } else if(windowWidth > 960 && mode !== 'm') {
    mode = 'm';
    redraw(mode);
  } else if(windowWidth > 640 && mode !== 'sm') {
    mode = 'sm';
    redraw(mode);
  } else if(mode !== 'default'){
    mode = 'default';
    redraw(mode);
  }
}
```

The variable `mode` is exposed outside of the `handleResize` function so that it isn't squashed every time a resize is called. As the visualization is loaded, and then every time a window resize event is captured, the code does not immediately attempt a redraw, but instead first checks whether the window's width (in this case—because it could be height, DPI, touch capability, etc.) should change the mode from what it currently is.

If so, it changes it, then redraws. Within `redraw`, you now can optionally pass this mode as a parameter in order to skip parts of the code that would only be relevant to other view modes—for example, resizing text that is hidden on a smaller screen.

Grouping

There is a client-side layer of representation grouping that doesn't rely on a full mode-backed redraw. Yes, you certainly should do everything that was just outlined to pull different sets in different viewing modes, but you can also optimize for all of the viewing sizes within those modes.

In the following code example, `data_visible` is some simple math determining how many items in your set are attempting to fit into one pixel.

Then `dataset_grouped` is a downsampling, per this number, of every `data_visible`-th element in the greater data set. The new filtered data set is much smaller, but it still shows *nearly* the same chart as the more granular one.

```
var data_visible = dataset.length / width;  
var dataset_grouped = dataset.filter(  
  function(d, i) { return i % Math.ceil(data_visible) == 0; }  
);
```

Figure 10-18 shows stock data without grouping. Notice how thick and jagged the line looks. Figure 10-19 shows it after grouping using the method just described. Although precision is technically lost, it's not recognizable precision, and it's a small trade-off for the benefits of reduced rendering complexity, data set size, and performance overhead.

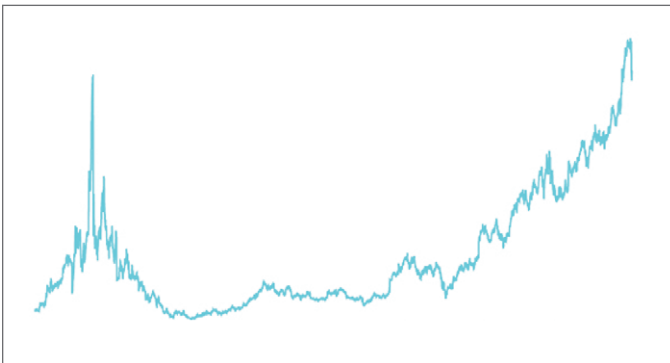


FIGURE 10-18

Ungrouped data. Notice the oddly thick and jagged line.

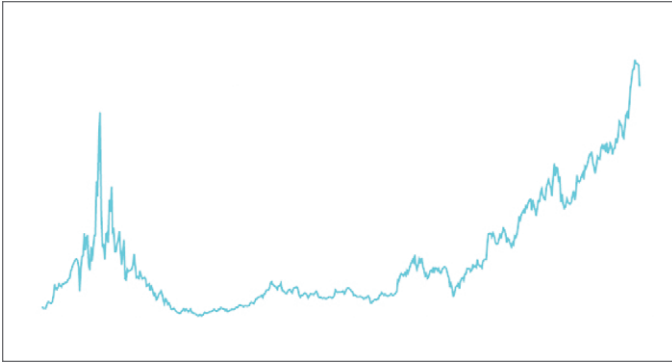


FIGURE 10-19

Grouped data. Not only does it look slightly better, it is much less taxing on the browser.

Data Responsiveness

At this point, you have a chart that will dynamically resize and redraw to fit its container. If your goal was to make something responsive enough to always fit, you've made it.

But you can make it much better.

With all the effort you've expended to creating a robust, lightning-fast API, you better use it. If at your default (smallest) screen size, you had room for, say, 320 data points on a line graph, but you have a total of 32,000, then on a 1,920px-wide screen, you should pull in more granular data.

Redrawing at Breakpoints

This is where passing a mode into your `redraw` function becomes important. In the preceding example, all you did was *redraw the exact same elements*. However, there is no mandate that you have to keep things uniform on a different screen size. As long as the message being conveyed by the data is the same, and the point you're trying to prove is always present, you should prove it with as much firepower as you have available.

To that end, your display *mode* should support whichever RESTful end point you are hitting to retrieve your data. That way, when you cross over between modes, you aren't just growing and shrinking the same set, you are actually showing more meaningful information. The snippet within `redraw` could look something like this:

```
var data_uri = 'http://api.yourEndpoint.com/  
retrieve?data=stock&mode='+mode;
```

Because you are calling this `redraw` function only when you are passing between states, and not every single time the page resizes, you don't run the risk of unnecessarily overloading your API. From there, you populate your data set from the JSON retrieved here, and `enter` all the new mode's data while `exit`-ing the old.

One such way to do that incredibly cheaply is to embed the mode within your data, and use it in your key function. That way, when the new data is selected, D3 recognizes that there is no overlap in elements, and will swap out the entire sets.

Redrawing at Interaction Points

Mode doesn't need to deal with viewport size only. Remember the concept of layering from your responsive data visualization tenets. Layering is the prioritization of the most crucial and digestible information, with an obvious indication that more granularity is available with interaction (think of the fitness app on the Apple Watch).

The idea behind layering is to present a rational default but enable users to dig into more complex or specific layers of data when the device's capabilities limit the presentation of both at the same time. Mode can handle this too. Instead of just being a string that represents screen size buckets, turn mode into an object instead:

```
var mode = {  
  'screen_size': 'default',  
  'a_layer': 0,  
  'b_layer': 1,  
  ...  
}
```

Mode now becomes the metadata on which you can base what sort of redrawing is required. In the preceding case, for example, the user is on a default (mobile-size)

screen, and has dug into set B's layering one level deep. The code to draw that layer's data simply updates `mode`, as it is global to the visualization:

```
function layer_b(status) {  
  //Do whatever drawing this needs  
  if(status === true) { //Dug in  
    mode['b'] += 1;  
  } else { //Exited  
    mode['b'] -= 1;  
  }  
}
```

Redraw can handle this accordingly, as it did with screen size.

The Bigger Picture

The fact remains: Neither unknowns on the data (input) or user (output) will ever happen without the other. A user with a particularly small screen size will interact with a different level of data granularity than a user on a touch-enabled television.

Creating visualization modes abstracts away from what is actually causing a change, and simplifies the problem into a set of inputs (the attributes of the mode object) and outputs (the specific ways in which the chart is redrawn in response).

Rational Defaults

Finally, you hopefully noticed that making our visualization rely on JavaScript completely breaks our ongoing adherence to progressive enhancement.

Remember that per progressive enhancement, JavaScript is at the very top layer, and is, as much as possible, relied on to create complex interaction, but not presentation.

While D3 is a JavaScript library and you are relying on pulling information via AJAX, that is not at all your only option. Throughout this and the previous chapter, you have been replacing things like the following:

```
<div id="viz"></div>
```

What this has (falsely) assumed is that there is no server-side representation of the data on the page to begin with.

In the most basic case, you can simply spit out the data you will be visualizing. After all, you have it. Suppose your replaceable element is still the following:

```
<div id="viz"></div>
```

Then there is nothing stopping you from filling it like so:

```
<div id="viz">
  <table>
    <thead>
      <tr>
        <td class="empty"></td>
        <th>ID</th>
        <th>Val</th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <th>Aldo</th>
        <td>341</td>
        <td>25</td>
      </tr>
      <tr>
        <th>Amber</th>
        <td>102</td>
        <td>105</td>
      </tr>
      <tr>
        <th>Bertram</th>
        <td>499</td>
        <td>6</td>
      </tr>
    </tbody>
  </table>
</div>
```

Now, users without JavaScript, search engine crawlers, and even users whose connections are slow enough that the AJAX call fails, are not missing a single bit of data.

What's more, you'll often *want* this to be your most basic case, as a table like this is:

- Easily scanned on tiny screens

- Easily parsed by screen-readers and search engine crawlers
- Much faster than any other implementation

I can't stress this point enough: Clear is always better than clever, and when client resources are limited, nothing is more clear than actually showing the data.

That being said...

Drawing on the Server

You don't strictly need D3.js to create SVG visualizations, right? Sure, it's responsible for the interaction and responsive handling of data and screen size changes, but what about initial page load?

The Long, Slow, Stupid Way

It's just markup that has been transformed by certain data points; you could craft each point by hand if you wanted. Say you had the following data set:

```
var dataset = [10,2,4,5.5];
```

You could, on the server, recognize that your largest element is 10, and call that 100%. Then make the rest 20%, 40%, and 55%. Your code could be as simple as this:

```
.bar {  
  background: #000;  
  content: "";  
  height: 12px;  
  margin-bottom: 2px;  
}  
<div class="bar" style="width:100%;"></div>  
<div class="bar" style="width:20%;"></div>  
<div class="bar" style="width:40%;"></div>  
<div class="bar" style="width:55%;"></div>
```

As shown in Figure 10-20, it looks just like a D3-backed visualization would look; and in fact, you could have even just hand-crafted an SVG. It's fine for an example that simple. Once you start working with more complex data, however, you would

quickly exceed the realm in which it's feasible to craft a markup representation of your data.



FIGURE 10-20

A bar graph created manually

D3.js on the Server

Thankfully, you can continue to rely on D3.js on the server. Yes, you read that right—you're going to be running JavaScript to create SVG visualizations on the server. If you're familiar with NodeJS, this shouldn't be a crazy concept. If not, here's the gist:

NodeJS is an open-source, cross-platform runtime environment for server-side and networking applications. Node.js applications are written in JavaScript. It contains a built-in library to enable applications to act as a web server without software such as Apache HTTP Server or IIS, and uses JavaScript collections of contained JavaScript files, called modules, that handle functionality.

Basically, you can run (properly formatted) JavaScript as server-side code. Radical.

The following example omits the interactive parts—this is because all you really care about for initial page render is the default rendered SVG. This example requires some knowledge of NodeJS and Node Package Manager, or at least a suspension of disbelief for my hand-wave-y explanations about how they work.

First, you'll need to pull in your two requirements: D3 and jsdom (github.com/tmpvar/jsdom). You certainly know what D3 is, but jsdom is a JavaScript implementation of the WHATWG (Web Hypertext Application Technology Working Group) DOM and HTML standards. That is to say, it's an implementation of the DOM in 100% JavaScript.

It's designed specifically for NodeJS, and in addition to the example presented here, it is used to simulate a DOM for creating server-side cached versions of pages for search engine crawlers.

Sorry, I'm gushing. Back to the code. In the terminal, using NPM, you install your two dependencies with the following:

```
npm install d3 jsdom
```

Cool, dependencies taken care of. Suppose you have the following data set:

```
var dataset = [
  { 'id': 1, 'created_at': 'Fri May 01 2015', 'val': 100},
  { 'id': 2, 'created_at': 'Mon Jun 15 2015', 'val': 300},
  { 'id': 3, 'created_at': 'Mon Jun 01 2015', 'val': 900},
  { 'id': 4, 'created_at': 'Fri May 15 2015', 'val': 2700},
  { 'id': 5, 'created_at': 'Wed Jul 01 2015', 'val': 8100}
];
```

Suppose further that you're making a simple scatter plot with the following D3.js code, which should be straightforward enough to understand by now:

```
var data = json_data.slice()
var format = d3.time.format('%a %b %d %Y')
var computeVal = function(d) { return d.val }
var computeDate = function(d) { return format.parse(d.created_at) }

var x = d3.time.scale()
  .range([8, 304])
  .domain(d3.extent(data, computeDate))

var y = d3.scale.linear()
  .range([144, 8])
  .domain(d3.extent(data, computeVal))

var svg = d3.select('#viz').append('svg:svg')
  .attr('width', 320)
  .attr('height', 160)

svg.selectAll('circle').data(data).enter()
  .append('svg:circle')
  .attr('r', 4)
  .attr('cx', function(d) { return x(computeDate(d)) })
  .attr('cy', function(d) { return y(computeVal(d)) })
```

Stash that away for a moment, because that's not actually the starting point. Instead, look at the following code:

```
var d3 = require('d3');
var jsdom = require('jsdom');
var html_stub = '<html><head></head><body><div id="viz"></div><script
src="js/d3.js"></script></body></html>';
```

To start, you're simply requiring D3 and jsdom. On line 3, you're creating the actual DOM that you'll be manipulating:

```
jsdom.env({ features : { QuerySelector : true }, html : html_stub
, done : function(errors, window) {
    // Code to generate and display visualization
}
});
```

Next, you're taking that DOM and passing it to jsdom. Note that on line 3, I've stubbed out where the D3 action will take place. At this point, the code creates an HTML document on the server, keeping it in memory for now. What you have in this document is as follows:

- The HTML base
- Something to replace with your visualization
- Script to load D3

To start manipulating the DOM, you'll use that `querySelector` function, provided by jsdom. It works just like jQuery's `$`. That's shown in the following code:

```
var el = window.document.querySelector('#viz');
var body = window.document.querySelector('body');
```

From there, you'll now write the D3 code you would expect. What's neat is that this is just generating the SVG on the server. The code is shown here:

```
var data = json_data.slice()
var format = d3.time.format('%a %b %d %Y')
var computeVal = function(d) { return d.val }
var computeDate = function(d) { return format.parse(d.created_at) }

var x = d3.time.scale()
    .range([8, 304])
```

```

    .domain(d3.extent(data, computeDate))

var y = d3.scale.linear()
    .range([144, 8])
    .domain(d3.extent(data, computeVal))

var svg = d3.select('#viz').append('svg:svg')
    .attr('width', 320)
    .attr('height', 160)

svg.selectAll('circle').data(data).enter()
    .append('svg:circle')
    .attr('r', 4)
    .attr('cx', function(d) { return x(computeDate(d)) })
    .attr('cy', function(d) { return y(computeVal(d)) })

```

Here's the problem: That D3 code isn't going to be present on the HTML page now present on your server. All that's there is the output—a totally accurate but totally static SVG representation of your data.

On the client, you would then need to select your visualization and create your data, enter, and exit selectors to handle interaction as you have done normally up until this point; but the step before this is actually moving the HTML file from memory to the server. That is done in the following way:

```

var fs = require('fs');
var svg_source = window.document.innerHTML

fs.writeFile('index.html', svg_source, function(err) {
    if(err) {
        console.log('Error saving document:', err);
    } else {
        console.log('The file was saved.');
```

In the preceding code, `fs` is simply a NodeJS file system. It allows for the storage and modification of files. The beauty of this code is that it enables you to store simply the markup for the SVG, serve a whole surrounding page, or send the output directly from NodeJS if the rest of your app is working within that system.

The outputted file is now displayed as shown in Figure 10-21, and the markup is exactly the same as it would have been if you had generated it client side.

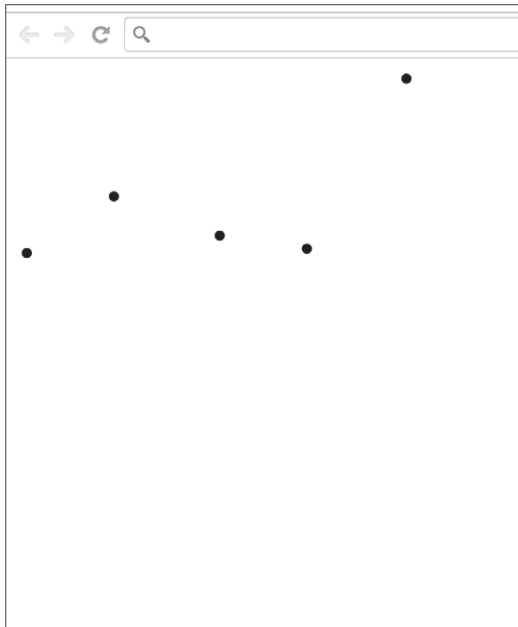


FIGURE 10-21

The server-side generated D3 SVG visualization

```
<html>
  <head>
  </head>
  <body>
    <div id="viz">
      <svg width="320" height="160">
        <circle r="4" cx="8" cy="144"></circle>
        <circle r="4" cx="226.36065573770495" cy="140.6"></circle>
        <circle r="4" cx="158.42622950819674" cy="130.4"></circle>
        <circle r="4" cx="75.93442622950819" cy="99.8"></circle>
        <circle r="4" cx="304" cy="8"></circle>
      </svg>
    </div>
    <script src="js/d3.js"></script>
  </body>
</html>
```

Caring about Performance

Running D3.js-based visualizations on the client is (as you hopefully now believe after I've beaten you over the head with it) great for manipulating the DOM. It isn't, however, great at being 100% lightning-fast when your data becomes large and complicated.

Intuitively, that makes sense. Just as adding content to an entire page complicates it and slows it down, adding data (content) to a layout slows it down. On top of that, DOM manipulations, using D3 or otherwise, are performance-expensive to begin with.

For an initial draw, there is absolutely no reason to do all of this work on the client if you can avoid it. Your result will be exactly the same, whether you create your initial view on the server or on the client.

The overhead of computing the layout, adding each DOM node, and correctly sizing and positioning it to represent your bound data can all be done on the server, rather than on a user's device.

Most importantly, creating your initial visualization on the server enables your progressive enhancement technique to begin in a much more *enhanced* state. Keeping in mind that SVG is extremely well supported, if you have the opportunity to serve users an experience that's nearly your beautiful interactive goal state to begin with, you certainly should.

Starting with a table is one (great) thing, but starting with a visualization that is virtually identical to what you'll be enhancing and optimizing for your best devices is ideal.

Driving Design with Data

When you are designing a landing page, you don't think: "I want to use a 12-column grid, and 36pt headers, and have five banded sections." You think: "I have this content, how can I make it work?"

That's no different here.

For every data set, there are visualization styles that lend themselves to it better or worse. You can't begin a project with the intention of making a donut graph just because.

Clear, Not Clever

Regardless of screen size, input type, data complexity, or bandwidth, the most important thing your visualization can be is *clear*. The incredibly powerful toolset you have at your disposal can also be a dangerous one: You don't want to waste it by making exciting, flashy, complicated visualizations that resize and layer and group unnecessarily.

Hopefully, most of the time the answer is yes. The brain is wired to consume data in a visual way, and it remembers a picture better than a word. Moreover, your data is likely complicated enough that it won't just be one number anyway.



TO THE POINT

The question you should always be asking yourself is this: Is this visualization clearer or more meaningful than a number?

Nonetheless, it's a good question to ask when you feel like you're pushing too far in the direction of visualizing for the sake of visualizing.

Summary

Leaving this chapter, you now have the toolset to build code-driven visualizations using D3.js. More importantly, you have the techniques and understanding to build them in such a way that responsiveness is inherent in their drawing methods. Your visualizations are now built in a way that responds to both data complexity and user device features.

You’ve also learned how to align these visualizations with the overarching mindset of progressive enhancement by creating rational defaults on page load, either by displaying the simplified data, or by running D3.js on the server to create SVGs.

Bringing that all together, you connected code-driven responsive visualization with the concepts of building API-first, loading only what you need to, and using lightweight, semantic markup in order to visualize responsibly.

Try It

- You’re going to do a responsive retrofit on a visualization. It’s a listing of a company’s stock price as a series of bar graphs for each day. If you remember from the beginning of this book, that means you’re taking an existing desktop site and using responsive techniques to make it work *at least pretty well* on smaller screens. The page, JS, and the data set are available as “Exercise 1” at responsivedatavisualization.com/chapter-ten.

Try going at it with the following techniques:

1. Only resize the visualization once. Then, instead of focusing on the D3-created visualization, make the rest of the page responsive.
 2. Now layer in responsive data visualization techniques using the repainting methods you learned in this chapter.
- “Exercise 2” in responsivedatavisualization.com/chapter-ten has a client-responsive solution to exercise one, but with the addition of a button that currently swaps out a much less populated data set. Do the work to make this solution responsive to both the user and the data set.
 - If you’re feeling particularly ambitious, the “Exercise 3” folder has simply the data sets and a boilerplate empty HTML page with D3 and jQuery imported. Go nuts, and share your solutions on the github project.



11

Building the Future-Friendly Web

It's time to move away from implementation details and look at the bigger picture. Responsive design is not just a solution for the current state of devices, but a way to build proactively—anticipating the likely needs of future users in order to provide the best experience. This chapter ties the knowledge you've gained throughout this book to a more theoretical discussion about just how you should think about building web experiences.

The Future of Data Design

At this point, you should be comfortable designing for the devices currently on the market. Recall or revisit Chapter 2 and Chapter 7, describing the tenets that guide responsive design and data visualization. Responsive design's guiding tenets are measure, flow, nesting, end points, breakpoints, typography, and images. Responsive data visualization's tenets are measure, flow, end points, breakpoints, images, grouping, and layering.

The tenets unique to responsive design are nesting and typography. Intuitively, this makes sense: Nesting layout design certainly has an effect on visualization, but it is more a function of the container than the data itself. The same can be said of typography: Responsive type design is not limited to data visualization, but rather will affect all aspects of your product.

The tenets unique to responsive data visualization are grouping and layering, which are closely tied to the changing amounts of data you both:

- Receive on the input side
- Can display in the output context

The big secret is that the two are completely intertwined. There is absolutely no way to think about the responsive data visualization of a project without looking at the greater whole of responsive design. Your end users never need to know the term *responsive* (or *design*, or *data*, or *visualization* for that matter). All they care about is ease of access to your easy to understand content in their own context.

How, then, can these two topics help guide what you'll be building for the screens that haven't been invented yet? To determine that, it's important to think first about how your inputs are likely to grow.

Growing Stockpiles of Data

The GovLab index at NYU (see <http://thegovlab.org/govlab-index-the-digital-universe/>) houses some amazing facts about data consumption:

- In 2012, the entire digital universe consisted of 2.7 zetabytes of data.
- As of April 2011, the U.S. Library of Congress alone housed over 235 terabytes of data. (If you don't know your byte-prefixes, one zetabyte is one billion terabytes, and one terabyte is 1,000 gigabytes.)
- In that same year, 2.5 exabytes (one exabyte equals one billion gigabytes) worth of data was generated every single day. The giant retailer Walmart collected 2.5 petabytes of data every hour (one petabyte equals 1,000 terabytes).

It's safe to assume that these numbers will continue growing as bandwidth becomes cheaper, data collection becomes more universal, and the amount of time spent on web-connected devices essentially becomes 24/7.

In fact, way back in 2009, Americans survived on an average daily diet of 34 gigabytes of data every day (see <http://bits.blogs.nytimes.com/2009/12/09/the-american-diet-34-gigabytes-a-day/>). To put this in context, 2009 was two years after the iPhone was released, and one year before the Ethan Marcotte article coining the phrase “responsive web design.”

The rise in mobile connectivity since that time has made Internet access in the first world a 24-hour given. Most American adults even sleep with their phones (see www.pewinternet.org/files/old-media//Files/Reports/2010/PIP_Adults_Cellphones_Report_2010.pdf).

The lesson? Every day which passes means that more data is available to us—and that means you will need to do even more work to process it into a visualization that is both *effective* and *efficient*. As a content creator for the web, you need to recognize that the amount of available input surpasses a level that humans can manually parse. Unless you create worthwhile, understandable, accessible experiences, users will simply see your project as noise.

Setting a Standard for Data Markup

The bottom line is clear: Although you can't control the dizzying amounts of data out there, you can solve the problem of data complexity at the raw level.

The system for content and data management at National Public Radio (NPR) has become a shining beacon for those trying to navigate the murky problem of separating content and display. The system, called COPE (Create Once, Publish Everywhere), does exactly what it sounds like. Content is stored in the CMS in a language- and presentation-agnostic way. It's simply the clean, markup-free material that is eventually displayed to users.

Using Structured Content

NPR's system is a modular approach to creating portable and entirely dependency-free blocks of content that can be simultaneously sent to web browsers, apps, or any other means for reaching their content. NPR's own Daniel Jacobson has described the system's benefits as follows:

Content portability ensures that the content can actually live and thrive in all platforms to which it gets distributed (even those that do not yet exist).

DANIEL JACOBSON

In practice, this means your data should be stored in such a way that anything or anyone can reach it without requiring a certain display. More simply, it means creating a good API to reach your data.

This advice echoes the principles of creating a useful API described in Chapter 8, but it takes them a step further. Your API is useful not only in terms of how you are displaying everything today, but in terms of creating a more structured, global way to reach your data in ways not yet conceived.



NOTE

*Straight from the horse's mouth:
Read about NPR's COPE
system from a member of
their team's post at [www
.programmableweb.com/
news/cope-create-once
-publish-everywhere/
2009/10/13](http://www.programmableweb.com/news/cope-create-once-publish-everywhere/2009/10/13).*

Creating a Fluid Display

Referring back to NPR's COPE system: They never have to make a change to how their data is stored to account for the emergence of a new platform. Everything is backed by *identical* content, and lightweight, platform-specific views are built on top of it.

Moving to clean, presentation-agnostic data gives your product the best chance to at least be *viable* to a future platform, and possibly enables it to be completely well-formed without any additional work.

Embracing Uncertainty

Throughout the history of the web, there have been times of disruption and standardization. In some cases, different browser vendors raced to support different subsets of markup, while designers flocked to third-party solutions such as Flash. In other cases, technical leaders rallied for a standards-based way to create web experiences.

During these periods, curveballs like the growth of the mobile web, the Internet of things, and the race for the next new hardware have always, and will always, change the web landscape. As designers in this landscape, we need to embrace the fact that we are never going to build anything that is future-proof. It would be ludicrous to think that we could ever anticipate what's coming next. We can, however, become comfortable with the fact that we are uncertain about what the next great leap is, and build in such a way that our products are not stuck in trendy, proprietary solutions of the moment, or reacting to problems that we have already solved.

In essence: You need to build without assumptions—for a world in which your content will be delivered to an immersive VR headset and a 2G-connected Android smartphone and everything in between.

Where Complexity Lies Today

Complexity, now, comes from two sources: the variety of devices available in the marketplace, and the availability of tools you can use to make things. As the number of things connected to the web grows, so does the amount of work required to create accessible content.

The barrier to creating things on the web is no longer hosting costs, web service complexity, technical acumen, or even basic Internet connectivity. Anyone who wants something online can find a way to get it there.

We are in a period of rapid growth on both the input (our own code) and output (the clients' devices) fronts. Our existing tools were not created with the device landscape of today (or tomorrow) in mind. Unfortunately, that includes web browsers. The work being done on the browser side today to remain standards-compliant and perpetually updated has come a long way, but it requires buy-in from both content producers and end users.

Device Availability

The amount of network-connected devices, with screens of any size, or no screens at all, is going to keep increasing. That doesn't mean, however, that they're going to keep getting *better*. The ubiquity of the web platform is a boon to designers, because we don't need to repeat our content, but also a burden, as we need to create styles for screens we never imagined. This is stated succinctly by Marcos Lara:

HTML5 has delivered on the promise of write once, run anywhere. As your code begins to find its way into places you never expected it to run, you're going to come face to face with the issues around formatting for the very small, to the very large screens.

MARCOS LARA

<https://developers.facebook.com/blog/post/599>

High-end and technically astounding devices will continue to be invented and permeate the marketplace, but so will low-end consumer devices to get large parts of the world online for the first time. That is, there will be just as many, if not more, cheap and *just okay* devices as there are expensive and powerful devices.

The complexity arises from ensuring that your content is standards-compliant and at least working on all of these devices.

Availability of Toys

On the flip side of this, you have a ludicrous number of options regarding what to use to get your content online. If you wanted to, you could still build a visualization in Flash. The ubiquity of open development, although absolutely a good thing in pushing innovation, has created a noisy marketplace of solutions that range from useless to perfect.

The complexity here arises from judging whether a solution fulfills the following requirements:

- It works for you.
- It works for your users.
- It will continue to work for future users.

The Future-Friendly Manifesto

Thinking about these two sources of complexity, and trying to distill a solution seems daunting. Thinking in a way that will fulfill those requirement above becomes the ultimate goal.

Some recognized big-brains in the web realm decided to get together and establish a set of guiding principles that explain this way of thinking. Called the “Future-friendly Manifesto,” it offers the following problem statement:

Disruption will only accelerate. The quantity and diversity of connected devices—many of which we haven't imagined yet—will explode, as will the quantity and diversity of the people around the world who use them. Our existing standards, workflows, and infrastructure won't hold up. Today's onslaught of devices is already pushing them to the breaking point. They can't withstand what's ahead. Proprietary solutions will dominate at first. Innovation necessarily precedes standardization. Technologists will scramble to these solutions before realizing (yet again) that a standardized platform is needed to

maintain sanity. The standards process will be painfully slow. We will struggle with (and eventually agree upon) appropriate standards. During this period, the web will fall even further behind proprietary solutions.

THE FUTURE-FRIENDLY MANIFESTO

<http://futurefriendlyweb.com/>

This can be distilled into four unique parts: accelerated disruption, the decay of old standards, proprietary innovation, and the creation of new standards from this innovation. The following sections look at each one of these.

Accelerated Disruption

Only *reacting* to what is happening today will put our solutions further and further behind, and changes are outpacing our ability to keep up with them. The project you are building right now will be interacted with on a screen that doesn't exist right now.

More than just the devices, the diversity of people using those devices will grow, as barriers to web access diminish. This means that disruption will not only happen on the side of high-end experiences, but also on the low end, where people will begin accessing the web for the first time on more cost-efficient devices.

Decaying Standards

Clearly, the web today is not the one that existed 25, or even five, years ago, except for all the technical ways in which it is the same. The promise of the web to theoretically reach everyone on the planet cannot be realized with the methods *currently* in use for the types of devices and networks that we solve for now. As interaction methods change, our standards will become outdated, and will need to be revisited.

Proprietary Innovation

New solutions, more creative ways of tackling problems, and more efficient methods of web design or data display will never be handed down as an edict from the Supreme Ruler of Web Design. Individuals will always create solutions for their own problems first. Oftentimes, these solutions will conflict with one another.

This is alright: Competition like that will *breed* the standards that are adopted. Through individuals pushing boundaries in different ways, ultimately the boundaries will move.

Setting New Standards

Building off of the proprietary innovation, new standards will need to be effected. Think about the outset of responsive web design as a whole. Until it was *actually* a problem, nobody was solving for different sizes of device; and when they were, it was done in different ways. First mobile-only sites were made, or users were directed toward apps. We used RESS and device detection.

Only when responsive design took hold, and many people began to do *similar* things with it, did it ever become something akin to a standard practice. This will continue to happen as new devices spring up with new features that we need to solve for.

The Friendly Web

How can you actually start working toward becoming comfortable with these four tent poles? Anticipating the future but not strictly building toward one proprietary avenue of innovation is a way to build “friendly” web experiences—that is, those that will work not only with today’s popular browsers, but also with those emerging on the top and bottom of the market.

This movement, to build a friendly web, has gained enough ground that you may have seen it cropping up around the web. The future friendly web’s badge is shown in Figure 11-1.

The future-friendly way of thinking centers on five related ways to practically build for this ambiguous future:

- Laser focus
- Orbiting around data
- Creating universal content
- Building for device taxonomy
- Thinking beyond mobile

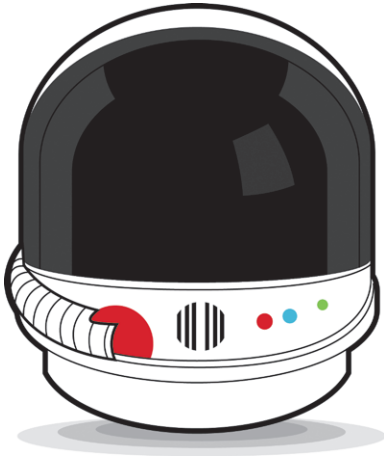


FIGURE 11-1

The future-friendly web badge

Laser Focus

As mentioned previously several times, you will never optimize everything for every device. Even if you were to manage every piece of complexity on the client side of your product at this very moment, you would once again be out of date tomorrow. Instead of focusing on devices, focus on what matters most to your end users: meaningful, intuitive content. Substance over style.

Orbit around Data

Remember NPR’s “Create Once, Publish Everywhere” system described earlier? This is that. A growing ecosystem of screens demands that the underlying data to be displayed enables flexible forms of access.

Define your base layer, your data, in a standards-focused way that promotes long-term integrity, has access points (an API) that allow meaningful and permanent references to all content, and supports writing and updating to stay relevant just as easily as it supports reading and consuming data.

Universal Content

Building off of the preceding, your data *must* be well-structured. This is not just for access; it builds into your actual visual design. You will never be building for a single

size of container ever again. Your content needs to be universally accessible so that you can, without massive expense, create new views that facilitate an efficient and understandable transfer of information from the underlying data.

Device Taxonomy

A phone with a screen in the camera will come out tomorrow, a watch with no screen will be available the next day, and by the weekend Instagram will be an ocular implant. You aren't solving for individual devices.

Building instead a *taxonomy* of device types enables you to optimize appropriately for a set of standardized features. Tools such as Detector (<http://detector.dmolson.com/>) enable you to abstract away from any one device, and react to the actual features available to that device. Detector is a PHP- and JavaScript-based browser and feature-detection library that can adapt to new devices and browsers on its own, without the need to pull from a central database of browser information. In essence, this is how RESS can help you build for new devices.

Beyond Mobile

Finally, remember that mobile is just the platform that brought us to the point of considering the issue of responsive web design. Enabling small, medium, and large screens to reach our content is a recent problem that we have just begun to feel comfortable solving. This will absolutely *not* be the last hurdle for web design.

Responsible Web Design

Bringing together all the concepts you've learned throughout this book, and now thinking about building things for an uncertain future, it's important to consider a different "r" word when approaching projects: responsible.

At least for the foreseeable future, everything you're going to build will be responsive. The whole point is to build something once that works everywhere. The line

you will now have to walk, knowing how to build responsive data visualization dashboards (or any other web project), is how to not fall into the traps of over-optimization, reliance on JavaScript for core functionality, and doing something just because it's cool.

You should always build with the mindset that you're trying to make a user's life easier. With that in mind, there are four tent poles supporting responsible responsive design and visualization.

Building API-first

The content of any blog page is simply words. You don't need to query any database to get access to those. Data visualization requires the extra step of actually needing...data...to...visualize. To create this most efficiently, you need to build for a machine with no screen as your base user, and ensure that you can get machine-parsable access to all of your data via its API.

A great API provides granular access, deep filtering, value-typed results, RESTful end points that come from real user intents, standardized formatting, “nice” errors, meticulous documentation, high performance, and high availability.

Not only will building API-first provide you with a presentation-agnostic base upon which you can build whatever your heart (or project manager) desires, but it also fundamentally changes the way in which your team works. Designers, frontend developers, and backend developers can agree to an API as a “handshake” and stub out the others' work in order to begin building in tandem.



NOTE

See Chapter 8 for details about what makes a good API, and how this affects a development team.

Respecting Progressive Enhancement

Hopefully you've picked up on the importance of progressive enhancement by now. Way back in Chapter 1, the extent of what we called “progressive enhancement” is illustrated in Figure 11-2. However, now you've learned what exactly your input(s) and output(s) are for data visualization, put into context in Figure 11-3.



FIGURE 11-2

The simple view of progressive enhancement you began with

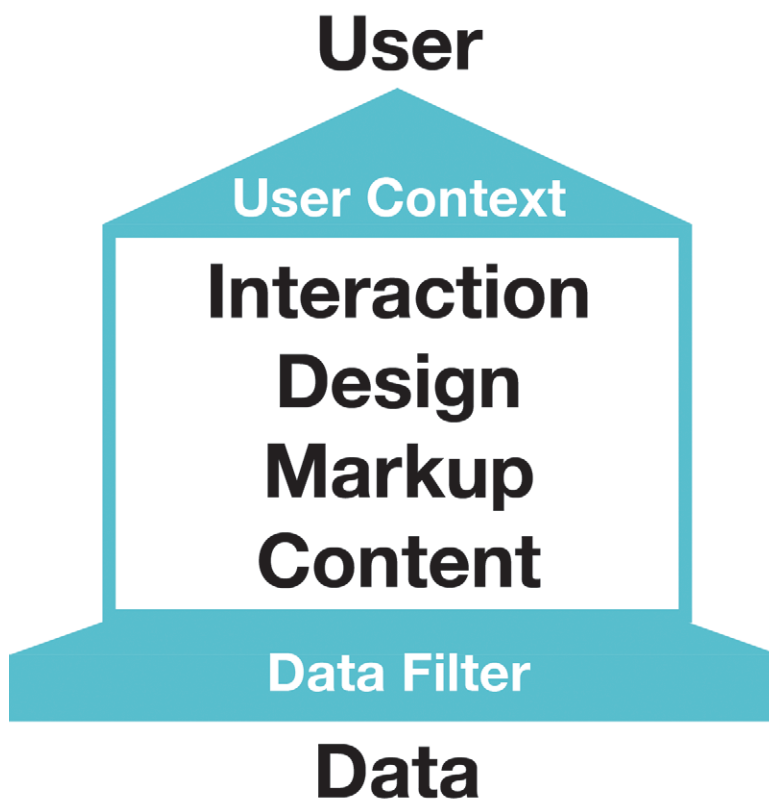


FIGURE 11-3

Progressive enhancement in the wild, with its additional inputs and outputs

There's no optional part of this. You *have* to build things this way if you plan to build things that will last. Device features will come and go, screen sizes will fall in and out of style; but the core makeup of the web: structured, semantic HTML, styled by CSS, and made interactive with JavaScript, is not going to change (or if it does, not nearly at the same pace), and this is the language you must speak.

Innovating

Don't wait for a library, a framework, or a toolkit to solve a problem for you. It will never be the perfect tool for your job. If you are building a presentation-agnostic API, then layering markup, design, and interaction on top of that, you should have a relatively open field of display options available to you. All you're taking in is numbers.

Remember that your visualization is never the *only* way to get to the meaning behind data, but it should be the most intuitive way to *reveal* that meaning. Data visualization, and the architecture behind it, should keep the following goals in mind:

- Identify
- Locate
- Manipulate
- Format
- Present

...data in such a way that communicating the *meaning* of the data becomes obvious and natural for a consumer. Keep the graphical display of it in mind. Whether it's a sentence explicitly communicating information, or an interactive 3D model, you need to maximize *efficiency* and *effectiveness*. That being said: Go nuts.

You're making someone's life easier by visualizing data on their screen, so find ways to communicate it in the best way possible.

Standardizing

When you do find a new way to do something effectively, see if you can repeat it. Then repeat it again. Then turn it into a pattern.

Following the future-friendly manifesto, your project-specific innovation is going to be what breeds standardization across the practice. As you find some beautiful way to structure your API end points, or generate well-formed and well-meta-tagged

markup, or connect your CSS to a bar graph, or handle rapid entry and exit of data, abstract it from one project into a solution that can be shared.

When you can standardize these solutions within your own projects, make it a point to share those standards with the community, whether via github, a blog post, or speaking at a conference.

Summary

You've learned the foundation of responsive web design, and seen how closely tied it is to responsive data visualization. Throughout this book, you have picked up skills that apply to building machine-parsable markup, designing universally with CSS, and adding interaction with JavaScript.

By this point it should be clear that the techniques used for design and visualization are nearly identical. You are given a set of inputs, and you need to create a clear, understandable output.

Regardless of the project, building this way has become known as “future-friendly,” reflecting an approach to building that pushes the web forward through rapid innovation and then meticulous standards creation. The basics of this way of thinking are simple:

- Acknowledge and embrace unpredictability.
- Think and behave in a future-friendly way.
- Help others do the same.

Future-friendliness describes design principles, not techniques. Trends in how things are created will grow, gain a spotlight, and then disappear. No single method, technology, or workflow is perfect for all projects.

If you begin to adopt a standard for your own data structure, starting at the API level, you can become an integral part of creating future-friendly solutions—first with your own projects, and then advancing the future-friendly web's handling of data as new devices, new screens, and new features come onto the market.

Close this book. Go build things.

THREE

PART

Additional Resources

APPENDIX: [Resources](#)

INDEX



A

Resources

You know those clickbait articles promising the “50 Best Web Design Resources,” which are quickly revealed to be selling some design service? The resources described here aren’t that. Rather than fill a book with a bunch of links, I’ve included the single best resource that I have found on each given topic, with a brief explanation of what makes it great.

Responsive Data Visualization

The resource: responsivedatavisualization.com

What makes it great: This is the website half of this book, so you've probably already familiarized yourself with it. Resources from previous chapters, example code, and exercises can all be found here, along with links to plenty of other resources not listed in this appendix.

Grids

The resource: thegridsystem.org

What makes it great: Not only is the site built on a grid, but it holds years' worth of articles on grid systems in design, on the web, and throughout history. Resources available on this site include Adobe program templates, CSS guides, downloadable prebuilt grids, and tutorials on how to build your own grid.

Infographic Design

The resource: coolinfographics.com

What makes it great: Randy Krum, the technical editor of this book, is the author of the Cool Infographics blog. It's *the source* for inspiration on infographic design, with a wealth of examples of data visualization and infographics from magazines, newspapers and the web.

Responsive Design

The resource: `bradfrost.github.io/this-is-responsive`

What makes it great: Brad Frost is one of the big-brains behind the future-friendly movement. A few years ago, he began to collect articles and simple abstractions of responsive methods on his own blog. Today, that collection lives at the preceding link, called “This Is Responsive.” The site features a huge number of responsive design patterns and downloadable modules, and links to articles about how to work, think, and build responsively.

D3.js

The resource: `d3js.org`

What makes it great: It just so happens that the place where you can download the source for D3.js is also a terrific resource for information about how you can effectively use it. The homepage of `d3js.org` details perfectly what the library is and how to use it. In addition, the site contains *tons* of example code, tutorials, and excellent documentation about how to best use D3.js.

More Resources Online

In the interest of not having you copy a bunch of text from a book, the resources mentioned in the chapters can be found at responsivedatavisualization.com. There you'll find all the exercises from this book, along with an expanded list of helpful resources on grids, data visualization, infographic design, responsive design, CSS, JavaScript, D3.js, API design, and the broader web design.

Index

A

<a> tag, 78
absolute minimum pixel width, 139
ACM (Association for Computing Machinery), 13
adaptive web design, 29
 adaptive layouts, 33–36
 device detection, 33
 responsive web design difference, 30
 RESS and, 30–33
adding data, `redraw()`, 302–305
 :after, 166
AJAX
 conditional loading, 168–170
 interaction, 239–240
 JavaScript linking, 171–172
 loading, 238–239
 performance, 170–171
anchor links, 158
Andreessen, Mark, 7

A (cont.)

API (application programming interface), 221–224
 availability, 276, 353
 content parity and, 231–232
 data
 filters, 271, 352
 granular access, 270, 351–352
 RESTful, 272–273, 352
 standardized format, 273, 352–353
 tables, normalized/denormalized, 272
 value typing, 271–272, 352
 documentation, 275–276
 error messages, 273–274
 fitness app, 279–282
 intents as URLs, 274–275
 JSON and, 224
 layering, visual layers, 283–284

A (cont.)

- open development*, 277–278
- performance*, 277
- versioning*, 276
- API key, 266
- apiURL, 223–224
- area charts, 202–203
- ARPANET, 4

B

- Backbone, 222
- bar charts, 197
- base size, 244
- Berners-Lee, Tim, 5
- big data, 191–192
- Bill, Max, 72
- Binh, Khoi, 75
- blind users, 266–267
 - bulk data*, 267
 - identifiers*, 268–269
 - open data*, 267
 - linked*, 269
 - structured data*, 267–268
- <blockquote>, 91
- <body> tag, 77
- Boulton, Mark, 75
- Braille media type, 120
- breakpoints, 59–62
 - data and*, 249–252
 - drawing modes*, 359
 - max-width*, 62–63
 - min-width*, 62–63

B (cont.)

- browsers
 - Chrome*, 7
 - Firefox*, 7
 - Internet Explorer*, 7
 - Mosaic*, 6–7
 - Netscape Navigator*, 7
 - Opera*, 7
 - Safari*, 7
- bubble charts, 203–204

C

- canvas, 255–256
- cartograms, 206–207
- charts. *See* graphs
- Chrome, 7
- CHSS (Cascading HTML Style Sheets), 103
- Cocktail Guide, 135–144
- commerce, mobile devices, 12–13
- communicating results
 - base forms*, 196
 - goals*, 195
 - graphical displays*, 195–196
 - graphs*
 - area charts*, 202–203
 - bar charts*, 197
 - bubble charts*, 203–204
 - cartograms*, 206–207
 - donut charts*, 199–200
 - Dorling Maps*, 203–204
 - heat maps*, 208–209

C (cont.)

- histograms*, 199
- line charts*, 201–202
- maps*, 206–207
- network diagrams*, 205–206
- pie charts*, 199–200
- radar charts*, 207–208
- scatter plots*, 201
- stacked bar charts*, 197–198
- tables*, 197
- tree maps*, 208
- Venn diagrams*, 204–205
- sentences*, 210
- timelines*, 210
- comparator, 244
- conditional loading, AJAX, 168–170
- consumer Internet, 6–7
- containers, nested, 55–56
- content parity, 230
 - API-first design and*, 231
- content-first design, 218–219
- correlation, 194
- CSS (Cascading Style Sheets), 22
 - CHSS (Cascading HTML Style Sheets)*, 103
 - content versus presentation*, 103
 - failure and*, 155
 - history*, 102–103
 - inheritance*, 109–110, 113
 - JavaScript and*, 164–166
 - media query*, 118–119
 - containers*, 125–126
 - features*, 122–123

C (cont.)

- media types*, 119–121
- retrofit responsive technique*, 126–127
- screen characteristics*, 121–124
- priorities*, 106–107
- reset stylesheet*, 111–112
- rules*, 105–106
- selectors*, 105–106
- source type priorities*, 333
- specificity*, 108–109

D

D3.js, 294–295

- data changes*, 301–308
- data function*, 296
- data reading*, 353–354
- domain*, 300
- enter function*, 296–301
- exit function*, 296–301
- importing*, 321–322
- JSON and*, 295–296, 298–301
- library*, 322–323
 - dynamic properties*, 324–325
 - selections*, 323–234
- range*, 300
- resources*, 393
- scale*, 344–346
- selectAll*, 336
- SVG output*, 326
- SVGs, creating*, 334–347
- transitions*, 346–347

D (cont.)

- data, 190–191
 - adding*, 301–302
 - `redraw()`, 302–305
 - big data*, 191–192
 - blind users*, 267–269
 - definition*, 188
 - dynamic*, 294
 - D3.js*, 294–295
 - data function*, 296
 - enter function*, 296–301
 - exit function*, 296–301
 - JSON*, 295–296
 - economy, performance*, 236–240
 - flexibility*
 - data sets*, 233–235
 - devices*, 232–233
 - in JavaScript*, 348–349
 - arrays*, 349–350
 - JSON*, 350
 - numbers*, 349
 - objects*, 350
 - strings*, 349
 - layering*
 - API response*, 257
 - design*, 258
 - interaction*, 258
 - markup*, 257–258
 - parsed data*, 257
 - processed data*, 256–257
 - raw data*, 256–257
 - machine-processable*, 268
 - metadata*, 192–193

D (cont.)

- quantitative messages*, 193–194
- removing*, 306–308
- responding to*, 258–259
- searchable*, 267
- sortable*, 267
- specificity*, 240–241
- structured*, 351
- data function, 296
- data science, 188–189
- data sources, 220–221
- data-content attribute, 170, 172
- default visualization methods, 211–212
- descriptive metadata, 193
- design
 - audience*, 259–260
 - content-first*, 218–219
 - desktop-first*, 36–37
 - mobile-first*, 37–38
- desktop-first design, 36–37
- desktop-focused design, 126–127
- deviation, 194
- device detection, adaptive web design
 - and, 33–36
- distribution, 194
- `<div>`, 105
- donut charts, 199–200
- Dorling Maps, 203–204
- dot-com bubble, 7
- downloads, 19–20
- dynamic data, 294
 - D3.js*, 294–295
 - data function*, 296

D (cont.)

- enter function*, 296–301
- exit function*, 296–301
- JSON*, 295–296

dynamic programming languages, 152

E

eBay, 12

economy principle

- data and, performance*, 236–240
- performance as design*, 46–47

embossed media type, 120

endpoints, 57–59

- data and*, 247–249

ENQUIRE, 5

enter function, 296–301

error message, blind users, 273–274

exit function, 296–301

F

failure, handling, 155

features, server-side detection, 164

Few, Stephen, 193–194

Firefox, 7

fitness app, 279–282

flexibility principle

- content*, 45–46
- data and*, 232–235

flexibility principles, devices, 43–45

flexible grids, 75–81, 87–88

- context*, 90–91
- negativity*, 91–92

F (cont.)

- padding*, 89, 92–93
- reusable*, 93–94
- columns*, 96
- gutter*, 95
- rows*, 96
- units*, 95
- X-count*, 95–96

flow, 51–54

- data and*, 245–247

fonts

- document base size and*, 113–114
- system*, 65–66
- target font*, 79
- web*, 65–66

G

geographic data, 194

getData, 311

graphs

- area charts*, 202–203
- bar charts*, 197
- stacked bar charts*, 197–198
- bubble charts*, 203–204
- cartograms*, 206–207
- donut charts*, 199–200
- Dorling Maps*, 203–204
- heat maps*, 208–209
- histograms*, 199
- interactive*, 218
- line charts*, 201–202
- maps*, 206–207

G (cont.)

- network diagrams*, 205–206
- pie charts*, 199–200
- radar charts*, 207–208
- scatter plots*, 201
- stacked area charts*, 202–203
- tables*, 197
- tree maps*, 208
- Venn diagrams*, 204–205
- grids, 72–73, 81–87
 - fixed*, 87
 - flexible*, 75–79, 87–88
 - reusable*, 93–96
 - gutters*, 89
 - outside-in build*, 88
 - padding*, 92–93
 - proportion*, 74–75
 - relative units*, 79–81
 - resources*, 392
 - responsive, abstracted sizes*, 147–148
 - rhythm*, 74
 - web design and*, 75
- grouping, 252–253, 309–311
 - client responsiveness and*, 360–361
- gutters, 89

H

- `<h1>`, 76, 78
- handheld media type, 120
- `handleResize` function, 359
- heat maps, 208–209
- histograms, 199

H (cont.)

- history of Internet
 - Andreessen, Mark*, 7
 - Berners-Lee, Tim*, 5
 - browsers*, 6–7
 - Chrome*, 7
 - Firefox*, 7
 - Internet Explorer*, 7
 - Netscape Navigator*, 7
 - Opera*, 7
 - Safari*, 7
 - consumer*, 6–7
 - dot-com bubble*, 7
 - ENQUIRE*, 5
 - HTML*, 5
 - HTTP*, 5
 - markup languages, mobile-related*, 9
 - mobile devices*, 8–10
 - commerce*, 12–13
 - usage*, 10–11
 - PCs, sales*, 7–8
 - TCP/IP protocols*, 5
 - World Wide web naming*, 5
- HTML (Hypertext Markup Language), 5
 - CHSS (Cascading HTML Style Sheets)*, 103
 - failure and*, 155
 - JSON and*, 295–296
 - tables*, 103–104
- HTTP (hypertext transfer protocol), 5
- human-readable markup, 226

I

identifiers, data, 268–269

images

canvas, 255–256

plugins, 254

rasters, 66–67

responsive, 68, 253–254

scripts, 254

SVGs, 254

desktop browser support, 255

meta support, 255

mobile support, 255

vectors, 66–67

infographics resources, 392

information, 191

inheritance, CSS and, 104, 109–110, 113

instant-api, 270

interactive charts, 218

Internet Explorer, 7

iPhone, 8–9

itemprop, 226

J

JavaScript, 22, 152. *See also* AJAX

CSS and, 164–166

D3.js, 294–295

defaults, 157–163

failure and, 155

interactivity and, 153–155

matchMedia object, 166–168

mobile-first design, 155–157

progressive enhancement and, 155–157

J (cont.)

responsive, 163–164

server-side feature detection, 164

jQuery, 295

JSON, 169, 183

API and, 224

HTML and, 295–296

list population, 228

just-in-time feature detection, 179–181

K

knowledge, 191

L

layering, 227–228

data

API response, 257

design, 258

interaction, 258

markup, 257–258

parsed, 257

processed, 256–257

raw, 256–257

visual layers, 283–284

layouts

adaptive, 33–36

responsive, 35–36

, 132–133, 225–226

libraries, D3.js, 324–325

line charts, 201–202

linked open data, 269

links, anchor links, 158

M

- machine-processable data, 268
- maps, 206–207
- Marcotte, Ethan, 26
- margins, 89
- markup
 - data and*, 257–258
 - meta information*, 258
 - visible*, 258
- markup languages
 - HTML (Hypertext Markup Language)*, 5
 - HTTP (hypertext transfer protocol)*, 5
 - human-readable markup*, 226
 - mobile-related*, 9
 - website-readable markup*, 226
- matchMedia object, 166–168
- max-width breakpoints, 62–63, 127
- measure, 50–51
 - data and*, 242–244
- media query (CSS), 118–119
 - containers*, 125–126
 - features*, 122–123
 - media types*, 119–121
 - retrofit responsive technique*, 126–127
 - screen characteristics*, 121–124
- MediaQueryList object, 167
- mediaQueryString, 167
- menus, anchor links, 158
- meta information, 225–226
- metadata, 192
 - descriptive*, 193
 - markup, content parity and*, 231–232
 - structural*, 192–193

M (cont.)

- min-width breakpoints, 62–63, 127
- .mobi, 19
- mobile devices, 8–10
 - commerce*, 12–13
 - markup languages*, 9
 - mobile tipping point*, 26
 - rendering complexity*, 128–129
 - usage*, 10–11
 - users*, 15
- mobile web
 - app download*, 19–20
 - bandwidth*, 15
 - benefits*, 15–17
 - CSS support*, 15, 22
 - design*, 17–20
 - JavaScript*, 15, 22
 - .mobi*, 19
 - m.websites*, 19
 - navigation*, 14
 - screen size*, 14
 - speed*, 14–15
- mobile-first design, 37–38
 - JavaScript and*, 155–157
- mobile-first method, 27
- Mosaic browser, 6–7
- Müller-Brockmann, Josef, 72
- m.websites, 19

N

- natural experiences, 212
- NCSA (National Center for Supercomputing Applications), 6–7

N (cont.)

nested Venn diagrams, 204–205
nesting, 54–57
Netscape Navigator, 7
network diagrams, 205–206
node-element binding, 306
nominal comparison, 194

O

Opera, 7
outside-in build, 88

P

padding, 89, 92–93
parity in content, 230
parsed data, 257

- content parity and*, 231–232

part-to-whole data, 194
PayPal, 12
PCs, sales, 7–8
perceived performance

- data and*, 236–237
- just-in-time feature detection*, 179–181
- train tracker*, 173–175
- visual feedback, loading and*, 175–178

performance

- actual*, 236
- AJAX
 - interaction*, 239–240
 - loading*, 238–239
- APIs, 277

P (cont.)

- perceived performance*, 172–173, 236–237
 - just-in-time feature detection*, 179–181
 - train tracker*, 173–175
 - visual feedback at loading*, 175–178

pie charts, 199–200
pixels, absolute minimum width, 139
plugins, 254
points of focus

- breakpoints*, 59–63
- data and*, 249–252
- endpoints*, 57–59
 - data and*, 247–249
- flow*, 51–54
 - data and*, 245–247
- grouping, data and*, 252–253
- images*, 66–68
 - data and*, 253–256
- layering, data and*, 256–258
- measure*, 50–51
 - data and*, 242–244
- nesting*, 54–57
- typography*, 63–66

principles of adaptive development

- economy*, 46–47
- flexibility*, 43–46
- specificity*, 47–49
- universality*, 39–42
 - content*, 39–40

print media type, 120
priorities, CSS, 106–107

P (cont.)

processed data, 256–257
programming languages
 dynamic, 152
 static, 152
progressive enhancement, 20–22, 37–38
 JavaScript and, 155–157
 sensible defaults, 233–234
projection media type, 120

Q

quantitative messages, 193–194

R

radar charts, 207–208
ranking data, 194
rasters, 66–67
 responsive, 68, 253–254
raw data, 256–257
 content parity and, 231–232
`redraw()` function, 302–305
rendering complexity, 128–129
resources
 D3.js, 393
 grids, 392
 infographics, 392
 responsive data visualization, 392
 responsive design, 393
responding to data, 258–259
responsive design
 grouping and, 360–361
 layouts, 101
 CSS, media query, 118–127

R (cont.)

fluidity, 125–126
 responsive grid, abstracted sizes,
 147–148
 reusable grid, 145–146
 resources, 393
responsive rasters, 68, 253–254
responsive web design, 26, 27
 clients and
 breakpoints, 358–359
 CSS and, 355
 resizing windows, 355–358
 data and
 breakpoints, 361–362
 interaction points, 362–363
 desktop-first design, 36–37
 every user, 28–29
 limited users, 27
 mobile-first design, 38
 RESS and, 30–33
 server and, 363–370
RESS (responsive design with server-side
 components), 30–33
retrofit responsive technique,
 126–127
reusable flexible grids, 93–94
 columns, 96
 gutter, 95
 rows, 96
 units, 95
 X-count, 95–96
Ruder, Emil, 72
rules, 105–106
RWD (responsive web design), 20

S

Safari, 7

scale, D3.js, 344–346

scatter plots, 201

schema.org, 226

screen media type, 120

scripts, 255

searchable data, 267

`selectAll`, 336

selectors, 105–106

semantic elements, 102

server-side feature detection, 164

sortable data, 267

sources of data, 220–221

specificity principle, 47

- CSS and*, 104, 108–109
- data and*, 240–241
- interface*, 49
- web as continuum*, 48

speech media type, 120

SSP (Stream-based Style Sheet Proposal), 103

stacked area charts, 202–203

stacked bar charts, 197–198

static programming languages, 152

structural metadata, 192–193

structured data, 351

- as content*, 221–224

stylistic elements, 102

SVG (Scalable Vector Graphics), 218–219, 254, 326–328

- `background-image`, 328
- `circle`, 330–331

S (cont.)

- D3.js and*, 294–295
 - creating SVGs*, 334–347
 - output*, 326
- desktop browser support*, 255
- `iframe`, 328–329
- `img`, 328
- `inline`, 329–330
- meta support*, 255
- mobile support*, 255
- `object`, 329
- squares*, 335–343
- static content and*, 330
- styling*, 330–334

system fonts, 65–66

T

tables, 103–104, 197

- semantics*, 104

target font, 79

time series, 194

`toggleClass` function, 161

transitions, D3.js, 346–347

tree maps, 208

Tschichold, Jas, 72

TTY media type, 120

TV media type, 120

typography

- fonts*
 - system*, 65–66
 - web*, 65–66
- readability, breakpoints*, 63–64

U

universality principle

content, 39–40

mobile web, 40–41

optimization, 41–42

support, 41–42

URI, data and, 230–231

user agent polling, 163–164

V

variables

dynamic programming languages, 152

static programming languages, 152

vectors, 66–67

Venn diagrams, 204–205

visible markup, content parity and,
231–232

V (cont.)

visual feedback, loading and, 175–178

visual hierarchy, 116

visual layers, 283–284

`<viz>`, 219

VML (Vector Markup Language), 218

W

web fonts, 65–66

website-readable markup, 226

World Wide Web, naming, 5

X

`XMLHttpRequest`, 169

Z

Zakas, Nicholas C., 13

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley's ebook EULA.