Practical Dojo Projects

Frank W. Zammetti
I've written four books now, including this one, and I've thanked a lot of people. From the usual suspects such as my wife, children, parents, and friends, to the unusual: fictional characters, various food, alien species, and even myself! At one point I said I had dedicated a book to everyone who matters, but it was my mother-in-law who pointed out I hadn't dedicated anything to her and thus hadn't dedicated a book to everyone who matters.

That got me to thinking. I've actually left out a few important people along the way, so let me try to cover all the bases this time around:


Oh yes . . . and my mother-in-law.

Now stop bugging me! ☺
## Contents at a Glance

About the Author ................................................................. xiii  
About the Technical Reviewer ....................................................... xv  
About the Illustrator .............................................................. xvii  
Acknowledgments ................................................................x ix x  
Introduction. .......................................................................... xxi

### PART 1  Setting the Table

| CHAPTER 1 | Introduction to Dojo | 3  |
| CHAPTER 2 | Dojo Core | 41  |
| CHAPTER 3 | Dijit | 115  |
| CHAPTER 4 | DojoX | 157  |

### PART 2  The Projects

| CHAPTER 5 | That Human Touch: Contact Manager and Client-Side Persistence | 215  |
| CHAPTER 6 | A Place for Your Stuff: Dojo Code Cabinet | 265  |
| CHAPTER 7 | Idiot Blob: The Game! | 329  |
| CHAPTER 8 | When the Yellow Pages Just Isn’t Cool Enough: Local Business Search | 379  |

INDEX .................................................................................. 441
PART 1 Setting the Table

CHAPTER 1 Introduction to Dojo

JavaScript: A Study in Evolution
Birth of a Language
Reasons for JavaScript’s Early Rise
Something Is Rotten in the State of JavaScript
The Unarguable Problems with JavaScript
As Ballmer Said: “Developers! Developers! Developers!…”
Standardization: The Beginning of Sanity
The Times They Are a Changin’: The Experienced
Come to Bear
What’s Old Is New Again: JavaScript in the Present
The White Knight Arrives: JavaScript Libraries
The Cream of the Crop: Introducing Dojo!
Let’s Take It from the Top: A Brief History of Dojo
The Philosophy of Dojo
The Holy Trinity: The Three Components of Dojo
Core
Dijit
DojoX
Dojo in Action: What Else? Hello World (Sort Of)
Getting Dojo onto the Page
Importing Style Sheets
Configuring Dojo
Importing Other Parts of Dojo
Finally: Dijits!
Getting Help and Information..............................................36
Summary........................................................................39

CHAPTER 2  Dojo Core....................................................41

Core vs. Base.....................................................................41
The Dojo “Include” Mechanism..........................................42
“Trimming the Fat”: Creating a Custom Dojo Build..............43
Come Fly with Me: Core in Detail......................................45
  dojo ......................................................46
  dojo.back..................................................90
  dojo.behavior...........................................92
  dojo.cldr.................................................94
  dojo.colors..............................................96
  dojo.currency..........................................96
  dojo.data...............................................98
  dojo.date...............................................98
  dojo.fx ................................................101
  dojo.i18n ..............................................104
  dojo.io ................................................105
  dojo.number..........................................107
  dojo.regexp...........................................110
  dojo.string............................................111
Summary........................................................................114

CHAPTER 3  Dijit.........................................................115

Dijit at a Glance............................................................115
  Getting Started with Dijit..............................................116
  Declarative vs. Programmatic Dijits..............................116
Common Dijit Methods and Attributes..............................118
Dijits Directly Under the Dijit Namespace..........................119
  ColorPalette.............................................120
  Dialog ....................................................121
  Editor ....................................................122
  InlineEditBox ..........................................123
  Menu, MenuItem, and MenuSeparator.........................125
  ProgressBar ............................................126
  TitlePane ...............................................127
  Toolbar ..................................................128
  Tooltip ..................................................130
  Tree .....................................................131
<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>dojox.gfx</td>
<td>179</td>
</tr>
<tr>
<td>Surface</td>
<td>182</td>
</tr>
<tr>
<td>Line</td>
<td>182</td>
</tr>
<tr>
<td>Polyline</td>
<td>183</td>
</tr>
<tr>
<td>Rect</td>
<td>183</td>
</tr>
<tr>
<td>Circle</td>
<td>183</td>
</tr>
<tr>
<td>Ellipse</td>
<td>184</td>
</tr>
<tr>
<td>Text</td>
<td>184</td>
</tr>
<tr>
<td>applyTransform()</td>
<td>184</td>
</tr>
<tr>
<td>dojox.grid</td>
<td>185</td>
</tr>
<tr>
<td>dojox.math</td>
<td>188</td>
</tr>
<tr>
<td>degreesToRadians() and radiansToDegrees()</td>
<td>190</td>
</tr>
<tr>
<td>distance()</td>
<td>190</td>
</tr>
<tr>
<td>factorial()</td>
<td>191</td>
</tr>
<tr>
<td>gaussian()</td>
<td>191</td>
</tr>
<tr>
<td>midpoint()</td>
<td>191</td>
</tr>
<tr>
<td>range()</td>
<td>192</td>
</tr>
<tr>
<td>sd(), or Standard Deviation</td>
<td>192</td>
</tr>
<tr>
<td>dojox.string</td>
<td>192</td>
</tr>
<tr>
<td>Builder</td>
<td>194</td>
</tr>
<tr>
<td>sprintf()</td>
<td>195</td>
</tr>
<tr>
<td>dojox.timing</td>
<td>195</td>
</tr>
<tr>
<td>Timer</td>
<td>195</td>
</tr>
<tr>
<td>Sequence</td>
<td>197</td>
</tr>
<tr>
<td>dojox.uuid</td>
<td>199</td>
</tr>
<tr>
<td>dojox.validate</td>
<td>202</td>
</tr>
<tr>
<td>isEmailAddress()</td>
<td>204</td>
</tr>
<tr>
<td>isInRange()</td>
<td>205</td>
</tr>
<tr>
<td>isIpAddress()</td>
<td>205</td>
</tr>
<tr>
<td>isNumberFormat()</td>
<td>205</td>
</tr>
<tr>
<td>isValidCreditCardNumber()</td>
<td>205</td>
</tr>
<tr>
<td>isValidIsbn()</td>
<td>206</td>
</tr>
<tr>
<td>isValidLuhn()</td>
<td>206</td>
</tr>
<tr>
<td>dojox.widget</td>
<td>206</td>
</tr>
<tr>
<td>Fisheye List</td>
<td>206</td>
</tr>
<tr>
<td>TimeSpinner</td>
<td>206</td>
</tr>
<tr>
<td>Toaster</td>
<td>208</td>
</tr>
<tr>
<td>Summary</td>
<td>211</td>
</tr>
</tbody>
</table>
PART 2  ■■■ The Projects

CHAPTER 5  That Human Touch: Contact Manager and Client-Side Persistence .............................................. 215

Requirements and Goals ............................................... 215
How We Will Pull It Off .............................................. 216
Dojo and Cookies ................................................ 217
The Dojo Storage System ........................................ 218
Gears .................................................. 220
Dissecting the Solution .............................................. 223
Writing styles.css .............................................. 226
Writing dojoStyles.css ........................................ 229
Writing index.htm ............................................... 230
Writing goodbye.htm .......................................... 240
Writing EventHandlers.js ..................................... 240
Writing Contact.js .......................................... 245
Writing ContactManager.js .................................. 250
WritingDataManager.js ....................................... 256
Suggested Exercises .............................................. 262
Summary .................................................. 263

CHAPTER 6  A Place for Your Stuff: Dojo Code Cabinet ........................................... 265

Requirements and Goals ............................................... 265
How We Will Pull It Off .............................................. 266
The dojo.data Namespace ....................................... 266
Gears Database Component ..................................... 268
Dissecting the Solution .............................................. 270
Writing index.htm ............................................... 271
Writing styles.css ............................................... 293
Writing CodeCabinet.js ....................................... 295
Suggested Exercises .............................................. 327
Summary .................................................. 328

CHAPTER 7  Idiot Blob: The Game! ........................................... 329

Requirements and Goals ............................................... 329
How We Will Pull It Off .............................................. 331
Revenge of the Demented Video Game Designer .......... 332
CHAPTER 8  When the Yellow Pages Just Isn’t Cool Enough: Local Business Search .......................... 379

Application Requirements and Goals .......................................................... 380
  The Yahoo APIs and the Dynamic <script> Tag Trick............................... 381
  JSON-P and Dojo ............................................................................. 384
  Yahoo Web Services Registration ....................................................... 385
  Yahoo Local Search Service ............................................................... 386
  Yahoo Maps Map Image Service ......................................................... 389

Dissecting the Solution ........................................................................... 391
  Writing index.htm ............................................................................ 392
  Writing styles.css ............................................................................ 408
  Writing App.js .................................................................................. 411

Suggested Exercises .................................................................................. 439
Summary .................................................................................................... 440

INDEX ........................................................................................................ 441
FRANK W. ZAMMETTI is a Sun worshipper who never really got over the cancellation of The Greatest American Hero in the ’80s. He is an avid indoorsman who spends his time contemplating all the possible ways he may one day depart this mortal coil involving tomatoes, giant foam fingers, and/or caterpillars. In addition, Frank is a developer of software applications for a large, global bank. He is a father, husband, and walker of the family dog. He lives, works, but mostly plays in the northeastern United States (although clearly not all of his mind inhabits that same location). Frank has now written four books, including this one, covering a wide range of topics from JavaScript, DOM Scripting, and Ajax to DWR (an Ajax library) to Ajax with Java technologies, and of course Dojo, which includes Ajax functionality. He continues to work on various world-changing technological advancements on the side that will likely be released as finished products by real companies some time next week, as is usually the case with every good idea he’s ever had. He is a contributor to a number of open source projects (even some that probably would prefer he wasn’t!) and is leader/founder of a couple, too.

Frank also rarely takes anything seriously, in case you hadn’t figured that out by now!
HERMAN VAN ROSMALEN works as a developer/software architect for De Nederlandsche Bank N.V., the central bank of the Netherlands. He has more than 20 years of experience in developing software applications in a variety of programming languages. Herman has been involved in building mainframe, PC, and client/server applications. Since 2000, however, he has been involved mainly in building all sorts of Java EE web-based applications. After working with Struts for years (pre-1.0), he got interested in Ajax and joined the Java Web Parts open source project in 2005. Herman also served as technical editor for the Apress titles *Practical Ajax Projects with Java Technology*, *Practical JavaScript, DOM Scripting, and Ajax Projects*; and *Practical DWR 2 Projects*. Herman lives in a small town, Pijnacker, in the Netherlands with his wife Liesbeth and their children, Barbara, Leonie, and Ramon. You can reach him via e-mail at herros@gmail.com.
ANTHONY VOLPE is the name of the dude who did the illustrations for this book. He has worked on several video games for various platforms with author Frank Zammetti, including Invasion: Trivia!, IO Lander, K&G Arcade, Spinsht, and Ajax Warrior. Anthony lives in Collegeville, Pennsylvania, and works as a graphic designer and front-end web developer. His hobbies include recording music, writing fiction, making video games, and going to karaoke bars to make a spectacle of himself. To check out all the madness, go here: www.planetvolpe.com. You’ll find that Anthony is a supremely multitalented guy and also a little . . . off-kilter . . . but in a very good way!
Acknowledgments

This is actually one of the few places in this book where I’m (err, mostly) serious, not my usual tongue-in-cheek self.

I quickly came to realize, upon writing the acknowledgments for my first book a few years ago, that a project of this nature doesn’t get done without a lot of help from a lot of folks. Sure, the author is effectively the engine driving things, but an engine without wheels, or a steering wheel, or headlights, or a real good stereo system doesn’t make a car on its own, and such is the case with writing a book.

So, I’d like to take this opportunity to thank all those who helped get this one out the door: Steve Anglin, editor, for getting this thing going in the first place. Sofia Marchant, project manager, who kept me on schedule throughout. Sharon Wilkey, copy editor, who made my gibberish (somewhat) coherent. Laura Cheu, production editor, for making it all look good. Herman van Rosmalen, technical reviewer extraordinaire, for making me pay attention to IE7. Anthony Volpe, illustrator, for saving me from having to draw stick figures colored with crayons. And also Joohn Choe, Jimi Browne, and Dominic Shakeshaft for making my experience with Apress what I’ve come to expect it to be: a very pleasant experience all around.

I’d also like to thank all the fine programmers contributing to the Dojo project itself. Not only did your efforts enable me to write this book (and by extension to make some extra money!), but you’ve also helped me get things done at my regular day job that would have otherwise taken months of effort to do myself. I think I can safely speak for everyone using Dojo when I say thank you very much; your efforts are most definitely appreciated.

Last but most definitely not least, I’d like to thank you, dear reader, for purchasing this book. It’s cliché for sports figures to say they couldn’t do it without the fans, but there’s truth in it, just as there’s truth in an author saying I couldn’t do it without the readers. You are very much appreciated by this author!

Oh yeah, I almost forgot: thank you to the Higgs boson. None of us could’ve done any of it without you! See you later!
Introduction

I’ve been developing software for a long time, by anyone’s count: over 25 years, about 15 or so of that “professionally” (meaning 15 years or so I’ve been getting paid to pretend I know what I’m doing). I’ve been doing web development specifically for something like 10 or so of those years. I can say, with all honesty, that I was ahead of the curve a number of times. I was writing what we all now call RIAs back around 1998 (two of the apps I wrote in that time frame are still in production, amazingly). I was one of those people who invented Ajax way back then, but didn’t think it was anything special and didn’t come up with a cute name for it, and hence I am not the one rich and famous for coining an acronym!

None of this is especially important, and I’m not even trying to impress you by saying any of it. I’m just setting the foundation for being able to honestly say this: I’ve seen a lot. I’ve done a lot of client-side development. I’ve seen the evolution from doing absolutely everything yourself, dealing with all the complexities and cross-browser issues, to where we are today, where you basically don’t write anything from scratch anymore, you find a good library (or combination of libraries) and proceed from there. This is the same evolution that every other programming language and/or platform has taken; it’s just the natural way of things. People over time figure out what works, encapsulate it in some generic, reusable code (a library), and everyone is the better for their efforts.

I’ve used a large number of libraries over the years too. Some have been very good; some have been very bad; some, of course, were somewhere in the middle. One of the biggest today also happens to be one of the best ever: Dojo.

However, Dojo in a way suffers from its own goodness: it’s a bit difficult to wrap your brain around. It’s capable of doing so much that understanding what it can do, and understanding how to use it, can be a challenge.

That’s why I’ve written this book. You see, I’ve used Dojo in a highly complex application on the job, and I’ve seen the power it offers, but I’ve also faced the challenge I’m talking about here. My hope is that this book saves you some of the effort I went through and gets you to the point of using Dojo to its full (and considerable) potential quickly.

I also wrote this book because I believe many developers want and really need a “practical” way to learn something like Dojo. We’ve all seen the articles on the Net that purport to teach you something but do so by showing you simplistic, contrived, limited examples. This sometimes works, but often you hear people saying, “I wish I had a real application so I could see what this article is talking about in a larger context.” Many developers like to have code they can rip apart and comprehend on their own, and that’s what I’m trying to offer with this book.

I think you’re going to like Dojo a lot. In fact, I know you will because it will make some very complex things amazingly simple. It will save you a ton of time and effort and enable you to impress your boss in short order! It is without question one of the most forward-looking libraries out there today. There are always very cool new features being explored in Dojo, stuff that you often wouldn’t even think is possible in a browser. It’s just plain fun to play with sometimes!
Now, if you like and have even half as much fun with this book as you do with Dojo, then we're in good shape . . . but that's my job, isn't it?

Overview of This Book

This book is divided into two parts. The first is more “introductory” in nature:

- Chapter 1 is an introduction to Dojo. We’ll start by looking at client-side development more generically, talk about JavaScript’s evolution a bit, and finally the evolution of libraries. We’ll look at Dojo’s philosophy, its overall structure, and some of what it offers.

- Chapter 2 is where we’ll delve into the first major component of Dojo: Core. We’ll see many of the functions it has to offer, and see examples of each. (Yes, these are small, contrived examples, contrary to what I’ve said elsewhere, but that’s what’s needed in this section.)

- Chapter 3 is a detailed look at Dijit, which is where the Dojo GUI widgets are housed.

- Chapter 4 takes a look at DojoX, where the eXperimental components in Dojo live (and, in my opinion, where some of the very coolest parts of Dojo can be found).

In Part 2 we’ll jump into the four applications:

- Chapter 5 is where we’ll build a contact management application that uses the Dojo storage system to store contacts client-side. We’ll get a look at our first dijit too, as well as an introduction to Google’s Gears project.

- Chapter 6 presents the code cabinet, an application where you can store code snippets, categorize them, and search for them. We’ll get into Dijit a great deal more and see Dojo’s data abstraction layer.

- Chapter 7 gives you a break from the more serious applications and shows you how to create a game using Dojo. You’ll see Dojo’s extensive graphics capabilities in action here.

- Chapter 8 is where you’ll find a mashup that uses some publicly available web services to look for businesses around a given area, and show a map of that business. You’ll be able to store favorites client-side with this application. You’ll see much more of Dijit, the storage system, and lots of Core functionality.

There’s lots to cover, but by the end I believe you’ll have a good feel for Dojo and will be able to leap ahead in your own projects with it.

Obtaining This Book’s Source Code

I remember the days when programming books came with a CD containing all the source code presented in the book. These days, with the advent of the Internet, publishers can frankly save some money, and ultimately generate more profit, by using the Internet to provide that source code, and that’s precisely the case with this book.
To obtain the source code for this book, hop on over to www.apress.com and click the Source Code/Download link. Find this book in the list and you'll be able to pull down a (fairly large) archive with all the code.

Please note that this book is written with the assumption that you have done this, and have the code ready to look at. Unfortunately, when writing a book you sometimes have to make compromises in the interest of keeping the page count down (and also of not having 50 pages of code in a row, which no one would want to read!). This compromise often means that some of the code isn't printed in the book, and such is the case here. I have a personal philosophy that you should be able to read a programming book in the bathroom, gross as that may seem! Therefore, I've tried very hard to show as much code as possible, and to always show the truly pertinent parts when I couldn't show something in its entirety. Still, it is assumed that you have the code, can (or already have) played with the applications when you begin each chapter, and can refer to the full code as needed. This will all enhance your ability to understand things as you progress through the chapters.

Besides that, the applications are fun to play with, so if for no other reason than that, download the source bundle and have it available before you go too much further.

**Obtaining Updates for This Book**

No expense has been spared to ensure the accuracy and correctness of everything in this book. We've sacrificed goats to the gods, paid off politicians, tossed salt over our shoulders, and otherwise worked very hard to do our best.

All of that likely won't matter, though, because we're human, and mistakes unfortunately happen.

Because of this, a current errata list is available on the home page for this book on the Apress web site. You can, and I thank you in advance if you do, submit errata for anything you find. Hopefully you won't need to, but few books of this nature get through unscathed. Everyone on the team that made this book happen has done their best to make this one of those few books, but you never know.

I personally apologize unreservedly in advance. I know that I'll rue the day mistakes are found. *(Rue the day? Who talks like that?)* If you find any, I'll buy you a car. I don't have a car; it's my dad's. But then again, didn't anyone ever tell you to make sure your optics are clean?

**Contacting the Author**

Like most authors, I despise people and like to interact with them as little as possible. I'm a hermit who lives beyond the Dune Sea, content to make Sand People run trembling in fear from my ability to throw weird sounds wherever I choose.

But, if you'd like to contact me anyway (perhaps you have some droids you believe may be mine?), you can contact me at fzammetti@omnytexas.com.

*(That was of course sarcastic. I don't despise people. Well, some of them, but that's for another book. Feel free to contact me about anything you want. Sending money would be better, but I'll take a “hello, how you doin’” too!)*
There are two ways to write error-free programs; only the third one works.
—Alan J. Perlis

Computer programming is tremendous fun. Like music, it is a skill that derives from an unknown blend of innate talent and constant practice. Like drawing, it can be shaped to a variety of ends—commercial, artistic, and pure entertainment. Programmers have a well-deserved reputation for working long hours but are rarely credited with being driven by creative fevers. Programmers talk about software development on weekends, vacations, and over meals not because they lack imagination, but because their imagination reveals worlds that others cannot see.
—Bruce Eckel and Larry O’Brien

No matter how slick the demo is in rehearsal, when you do it in front of a live audience the probability of a flawless presentation is inversely proportional to the number of people watching, raised to the power of the amount of money involved.
—Mark Gibbs

Most software today is very much like an Egyptian pyramid with millions of bricks piled on top of each other, with no structural integrity, but just done by brute force and thousands of slaves.
—Alan Kay

I would be delighted to offer any advice I can on understanding women. When I have some, I’ll let you know.
—Captain Jean-Luc Picard

There comes a time when you look into the mirror and realize that what you see is all that you will ever be. Then you accept it, or you kill yourself. Or you stop looking into mirrors.
—Londo Mollari
The evolution of client-side development has been a remarkable journey spread out over a very short period of time. From the early days, which were only ten years or so ago, to the present, we have seen a huge change in the way things are done, the expectations placed upon those doing the work, and the results they generate. No longer is it acceptable for someone lacking a real programming background to churn out code for a web page in his or her spare time. No, there's a certain degree of professionalism now asked of us developers, and it's our job to figure out how to deliver the required results in what is usually a very compressed time frame.

Fortunately, because of this evolution, delivering powerful, good-looking, professional-quality web-based applications is far easier than ever before. You are no longer on your own, required to write every last bit of code, or at least to find snippets here and there and cobble it all together into some sort of coherent whole. No, those days are, for the most part, long gone, and it's thanks to things like Dojo that I can say that.

Before we get to Dojo, though, and all the wonderfulness it brings us, let's look back for a little bit of a history lesson, because only then can you fully appreciate what Dojo brings to the table.

JavaScript: A Study in Evolution

In honor of Charles Darwin, let's now take a brief tour of JavaScript and how it's evolved over time. This is in no way meant to be a detailed history lesson, but I think it'll give you a good overview of the past decade or so of client-side web development involving JavaScript.

Birth of a Language

The year was 1995, and the Web was still very much in its infancy. It's fair to say that the vast majority of computer users couldn't tell you what a web site was at that point, and most developers couldn't build one without doing a lot of research and on-the-job learning. Microsoft was really just beginning to realize that the Internet was going to matter. And Google was still just a made-up term from an old *The Little Rascals* episode.1

---

1. The word *google* was first used in the 1927 Little Rascals silent film *Dog Heaven*, to refer to having a drink of water. See http://experts.about.com/e/g/go/Google.htm. Although this reference does not state it was the first use of the word, numerous other sources on the Web indicate it was. I wouldn't bet all my money on this if I ever made it to the finals of *Jeopardy*, but it should be good enough for polite party conversation!
Netscape ruled the roost at that point, with its Navigator browser as the primary method for most people to get on the Web. A new feature at the time, Java applets, was making people stand up and take notice. However, one of the things they were noticing was that Java wasn't as accessible to many developers as some (specifically, Sun Microsystems, the creator of Java) had hoped. Netscape needed something simpler and more accessible to the masses of developers it hoped to win over.

Enter Brendan Eich, formerly of MicroUnity Systems Engineering, a new hire at Netscape. Brendan was given the task of leading development of a new, simple, lightweight language for non-Java developers to use. Many of the growing legions of web developers, who often didn't have a full programming background, found Java's object-oriented nature, compilation requirements, and package and deployment requirements a little too much to tackle. Brendan quickly realized that to make a language accessible to these developers, he would need to make certain decisions, certain trade-offs. Among them, he decided that this new language should be loosely typed and very dynamic by virtue of it being interpreted.

The language he created was initially called LiveWire, but its name was pretty quickly changed to LiveScript, owing to its dynamic nature. However, as is all too often the case, some marketing drones got hold of it and decided to call it JavaScript, to ride the coattails of Java. This change was actually implemented before the end of the Navigator 2.0 beta cycle. So, for all intents and purposes, JavaScript was known as JavaScript from the beginning. At least the marketing folks were smart enough to get Sun involved. On December 4, 1995, both Netscape and Sun jointly announced JavaScript, terming it “complementary” to both Hypertext Markup Language (HTML) and Java (one of the initial reasons for its creation was to help web designers manipulate Java applets easier, so this actually made some sense). The shame of all this is that for years to come, JavaScript and Java would be continually confused on mailing lists, message forums, and in general by developers and the web-surfing public alike!

It didn't take long for JavaScript to become something of a phenomenon, although tellingly on its own, rather than in the context of controlling applets. Web designers were just beginning to take the formerly static Web and make it more dynamic, more reactive to the user, and more multimedia. People were starting to try to create interactive and sophisticated (relatively speaking) user interfaces, and JavaScript was seen as a way to do that. Seemingly simple things like swapping images on mouse events, which before then would have required a bulky browser plug-in of some sort, became commonplace. In fact, this single application of JavaScript, flipping images in response to user mouse events, was probably the most popular usage of JavaScript for a long time. Manipulating forms, and, most usually, validating them, was a close second in terms of early JavaScript usage. Document Object Model (DOM) manipulation took a little bit longer to catch on for the most part, mostly because the early DOM level 0, as it came to be known, was relatively simplistic, with form, link, and anchor manipulation as the primary goals.

---

2. As a historical aside, you might be interested to know that version 2.0 of Netscape Navigator introduced not one but two noteworthy features. Aside from JavaScript, frames were also introduced. Of course, one of these has gained popularity, while the other tends to be shunned by the web developer community at large, but that's a story for another book!
Reasons for JavaScript’s Early Rise

What made JavaScript so popular so fast? Probably most important was the very low barrier to entry. All you had to do was open any text editor, type in some code, save it, load that file in a browser, and it worked! You didn’t need to go through a compilation cycle or package and deploy it, none of that complex “programming” stuff. And no complicated integrated development environment (IDE) was involved. It was really just as easy as saving a quick note to yourself.

Tying in with the theme of a low barrier to entry was JavaScript’s apparent simplicity. You didn’t have to worry about data types, because it was (and still is) a loosely typed language. It wasn’t object-oriented, so you didn’t have to think about class hierarchies and the like. In fact, you didn’t even have to deal with functions if you didn’t want to (and wanted your script to execute immediately upon page loading). There was no multithreading to worry about or generic collections classes to learn. In fact, the intrinsic JavaScript objects were very limited, and thus quickly picked up by anyone with even just an inkling of programming ability. It was precisely this seeming simplicity that led to a great many of the early problems.

Something Is Rotten in the State of JavaScript

Unfortunately, JavaScript’s infancy wasn’t all roses by any stretch, as you can see in Figure 1-1. A number of highly publicized security flaws hurt its early reputation considerably. A flood of books aimed squarely at nonprogrammers had the effect of getting a lot of people involved in writing code who probably shouldn’t have been (at least, not as publicly as for a web site).

Figure 1-1. JavaScript: The ugly ducking of the programming world?
Probably the biggest problem, however, was the frankly elitist attitude of many “real” programmers. They saw JavaScript’s lack of development tools (IDEs, debuggers, and so on), its inability to be developed outside a browser (in some sort of test environment), and apparent simplicity as indications that it was a script-kiddie language, something that would be used only by amateurs, beginners, and hacks. For a long time, JavaScript was very much the ugly duckling of the programming world. It was the Christina Crawford3 of the programming world, forever being berated by her metaphorical mother, the “real” programmers of the world.

---

**Note**  Script-kiddie is a term from the hacker underground that refers to a hacker who has no real ability and instead relies on programs and existing scripts to perform his exploits. The term is now used in a more generic sense to describe someone who relies on the ingenuity of others to get things done—for example, coders who can Google real well and copy the code they find rather than writing it themselves. Make no mistake, copying existing code is generally a good thing, but someone who isn’t ultimately capable of writing the code if they had to is a script-kiddie.

---

The Unarguable Problems with JavaScript

Although it’s true that JavaScript wasn’t given a fair shake early on by programmers, some of their criticisms were, without question, true. JavaScript was far from perfect in its first few iterations, a fact I doubt that Netscape or Brendan Eich would dispute! As you’ll see, some of it was a simple consequence of being a new technology that needed a few revisions to get it right (the same problem Microsoft is so often accused of having), and some of it was, well, something else.

So, what were the issues that plagued early JavaScript? Several of them tend to stand out above the rest: browser incompatibilities, memory usage, and performance.

Browsers for a long time implemented JavaScript in a myriad of different ways. Very simple things such as methods of the String object would frequently not work the same way in one browser vs. another. (Anyone who remembers trying to do simple `substring()` calls on a string around the 1996 time frame will know all too well about this!) Today, the differences in JavaScript implementations are generally few and far between and tend to be rather easy to work around. That’s certainly not to say you won’t occasionally be burned by differences, but typically you’ll find that JavaScript itself is probably 99 percent or better compatible between all the major browsers. It’s usually differences in the DOM, which there are still plenty of, that can bite you from time to time.

---

3. Christina Crawford was the daughter of Joan Crawford, and her story is told in the classic movie *Mommy Dearest* ([www.imdb.com/title/tt0082766](http://www.imdb.com/title/tt0082766)). Even if you don’t remember the movie, you almost certainly remember the phrase “No more wire hangers!” uttered by Joan to Christina in what was probably the most memorable scene in the movie.
Memory usage with early implementations also left a lot to be desired. JavaScript, being an automatic garbage-collected language, leaves all the memory allocation/deallocation in the hands of the browser and the JavaScript interpreter. It used to be common to find objects gumming up the works long after they should have been destroyed. This is rightly considered an outright bug in the JavaScript engine, but thankfully these types of things rarely occur anymore. To be sure, you can have memory leaks in JavaScript, but they are a result of more-sophisticated techniques being employed; it’s rarely the engine itself misbehaving.

Performance was a big problem for a very long time, and even today continues to be an issue. In the early days, it was rare to see an arcade-type game written in JavaScript because the performance of the JavaScript engines was just nowhere near good enough for that. Today, games like that can be found with little problem. Performance problems definitely still exist, and worse still they aren’t always consistent. String concatenation is known to be much slower in Microsoft Internet Explorer today than in Mozilla Firefox, and it’s probably not even as fast as it could be in Firefox. Still, overall, performance is hugely improved over what it once was.

As Ballmer Said: “Developers! Developers! Developers!...”

All that being said, though, there was one true reason JavaScript wasn’t embraced by everyone from the get-go, and that’s developers themselves!

Early JavaScript developers discovered that they could do all sorts of whiz-bang tricks—from fading the background color of a page when it loaded to having a colorful trail follow the cursor around the page. You could see various types of scrolling text all over the place, as well as different page-transition effects, such as wipes and the like. Although some of these effects may look rather cool, they serve virtually no purpose other than as eye candy for the user. Now, don’t get me wrong here, eye candy is great! There’s nothing I like more than checking out a new screen saver or a new utility that adds effects to my Windows shell. It’s fun! But I always find myself removing those things later, not only because they hurt system performance, but also because they quickly become annoying and distracting.

Here’s a quick test: if you are using Microsoft Windows, take a look at the Performance options for your PC (accessed by right-clicking My Computer, selecting Properties, clicking the Advanced tab, and clicking the Settings button under the Performance group). Did you turn off the expanding and collapsing of windows when minimized and maximized? Did you turn off shadows under the cursor? Did you disable the growing and shrinking of taskbar buttons when applications close? Many of us make it a habit to turn off this stuff, not only because it makes our systems snappier (or at least gives that perception), but also because some of it just gets in the way. Seeing my windows fly down to the taskbar when I minimize them is pretty pointless. Now, you may argue that it depends on the implementation. On a Macintosh, the effects that come into play when windows are minimized and maximized are better and not as annoying—at least most people seem think that, and to a certain extent I agree. But you still have to ask yourself whether the effect is helping you get work done. Is it making you more productive? I dare say the answer is no for virtually anyone. So although there may be degrees of annoyance and obtrusiveness, certain things are still generally annoying, obtrusive, and pointless. Unfortunately, this is what the term Dynamic HTML (DHTML) means to many people, and while I wish it weren’t so, it isn’t at all an undeserved connotation to carry. Some people even go so far as to say DHTML is the Devil’s buzzword!
Early JavaScript developers were huge purveyors of such muck, and it got old pretty fast. I don’t think it is going too far to say that some people began to question whether the Web was worth it, based entirely on the perception that it was a playground and not something for serious business.\footnote{I remember a television commercial of a bunch of web developers showing their newly created site to their boss. The boss says there needs to be more flash, like a flaming logo. The developers look at him a little funny and proceed to put a flaming logo on the page. It was pretty obvious to anyone watching the commercial that the flaming logo served no useful purpose, and in fact, had the opposite effect as was intended in that it made the site look amateurish. It’s so easy to abuse eye candy that it’s not even funny!} A web site that annoys visitors with visual spam is not one they will likely use again. And if you’re trying to make a living with that site and your company’s revenues depend on it, that’s going to lead to bad news real fast!

This obviously was not a failing of the technology. Just because we have nuclear weapons doesn’t mean we should be flinging them all over the place! I suppose equating nuclear war to an annoying flashing thing on a web page is a bit of hyperbole, but the parallel is that just because a technology exists and enables you to do something doesn’t necessarily mean you should go off and do it.

\section*{Are Effects Just the Deanna Troi of the Enterprise Bridge Crew? (Read: Just for Looks)}

Let’s tackle one question that often comes to mind first: why do we need effects at all? Isn’t it just a bunch of superfluous eye candy that doesn’t serve much purpose other than to make people go “ooh” and “aah”? Well, first off, if you’ve ever designed an application for someone else, you know that presentation is an important part of the mix. The more people like how your application looks, the more they’ll like how it works, whether it works well or not. It’s a relative measure. That’s the lesser reason, although one that should not be quickly dismissed.

The much more important reason has do with how we perceive things. Look around you right now. Pick up any object you want and move it somewhere else. Did the object just pop out from the starting point and appear at the new location? No, of course not! It moved smoothly and deliberately from one place to another. Guess what? This is how the world works! And furthermore, this is how our brains are wired to expect things to work. When things don’t work that way, it’s jarring, confusing, and frustrating.

People use movement as a visual cue as to what’s going on. This is why modern operating systems are beginning to add all sorts of whiz-bang features, such as windows collapsing and expanding. They aren’t just eye candy. They do, in fact, serve a purpose: they help our brains maintain their focus where it should be and on what interests us.

In a web application, the same is true. If you can slide something out of view and something else into view, it tends to be more pleasant for the users, and more important, helps them be more productive by not making them lose focus for even a small measure of time.

And seriously, how exactly does the ship’s shrink manage to always be around on the bridge? Is she necessary personnel for running the Enterprise? I know, I know, she finally passed the bridge crew test and was promoted to commander in the episode “Thine Own Self,” but still, she was always floating around the bridge long before then. I suppose she was on the lookout for space madness or something, who knows.
So, when you hear \emph{DHTML}, don't automatically recoil in fear, as some do. This term still accurately describes what we're doing today from a purely technical definition. However, you should, at the same time, recognize that the term does have a well-earned negative connotation, brought on by the evils of early JavaScript developers, not the technology they were using. And at the end of the day, that was the underlying source of the problems people saw back then (and still today, although thankfully to a lesser extent).

Part of the evolution of the JavaScript developer was in starting to recognize when the super-cool, neat-o, whiz-bang eye candy should be put aside. Developers began to realize that what they were doing was actually counterproductive, because it was distracting and annoying in many cases. Instead, a wave of responsibility has been spreading over the past few years. Some will say this is the single most important part of JavaScript's overall evolution toward acceptance.

You can still find just as many nifty-keen effects out there today as in the past, perhaps even more so. But they tend to truly enhance the experience for the user. For example, with the yellow fade effect (originated by 37signals, \url{www.37signals.com}), changes on a page are highlighted briefly upon page reload and then quickly fade to their usual state. Spotting changes after a page reload is often difficult, and this technique helps focus users on those changes. It enhances users' ability to work effectively. This is the type of responsible eye candy that is in vogue today, and to virtually everyone, it is better than what came before.

\begin{quote}
\textbf{Note} To see an example of the positive usage of the yellow fade effect, take a peek at the contact form for Clearleft at \url{http://clearleft.com/contact}. Just click the Submit button without entering anything and see what happens. You can also see the effect all over the place in the 37signals Basecamp product, at \url{www.basecamphq.com/} (you'll need to sign up for a free account to play around). You can get a good sense of where and why this seemingly minor (and relatively simple technically) technique has gained a great deal of attention. Other 37signals products use this technique, too, so by all means explore; it's always good to learn from those near the top! And if you would like to go straight to the source, check Matthew Linderman's blog entry at \url{www.37signals.com/svn/archives/000558.php}.
\end{quote}

\section*{Standardization: The Beginning of Sanity}

In early 1996, shortly after its creation, JavaScript was submitted to the European Computer Manufacturers Association (ECMA) for standardization. ECMA (\url{www.ecma-international.org}) produced the specification called ECMAScript, which covered the core JavaScript syntax, and a subset of DOM level 0. ECMAScript still exists today, and most browsers implement that specification in one form or another. However, it is rare to hear people talk about ECMAScript in place of JavaScript. The name has simply stuck in the collective consciousness for too long to be replaced, but do be clear about it: they are describing the same thing, at least as far as the thought process of most developers goes!

\footnote{I'm not only the hair club president, but I'm also a client. I have some old web sites in my archives (thankfully, none are still live) with some really horrendous things on them! I certainly was not immune to the DHTML whiz-bang disease. I had my share of flaming logos, believe me. I like to think I've learned from my mistakes (and so would my boss).}
The reason this standardization is so important is because it finally gave the browser vendors a common target to shoot for, something they hadn't had before. It was no longer about Microsoft having its vision of what JavaScript should be, and Netscape of course knowing what it should be, having created it and all. Now it was about hitting a well-defined target, and only then building on top of it (think enhanced feature sets that one browser might provide that another might not). At least if the underlying language implementation was based on a common specification, we developers would no longer have to hit a moving target (well, to be fair, not quite as active a moving target anyway). Without standardization, it’s likely that JavaScript would never have become what it is today, wouldn't be used as much as it is today. Either that or we'd have settled on one browser over another just to make our lives as developers easier, and that's no more a desirable outcome than no JavaScript at all.

The Times They Are a Changin': The Experienced Come to Bear
After the initial wave of relatively inexperienced developers using JavaScript, and many times doing so poorly, the next iteration began to emerge. Certain common mistakes were recognized and began to be rectified.

Perhaps most important of all, the more-experienced programmers who had initially shunned JavaScript began to see its power and brought their talents to bear on it. Those with true computer science backgrounds began to take a look and point out the mistakes and ways to fix them. With that input came something akin to the Renaissance. Ideas began to flow, and improvements started to be made. It wasn't the final destination, but an important port of call along the way.

Although a lot of the early problems with JavaScript undoubtedly did come from less-experienced programmers getting into the mix, certainly that didn't account for everything. Overnight, thousands of otherwise good, experienced programmers got stupid all at once!

As I mentioned earlier, working on JavaScript was almost too easy in a sense: throw some code in a file, fire up a browser, and off you go! In most other languages, you have a compile cycle, which tends to ferret out a lot of problems. Then you often have static code-analysis tools, which find even more things to fix. You may even have a code formatter to enforce the appropriate coding standards. None of this is (typically) present when working with JavaScript. I put typically in parentheses because modern development tools now exist to give you all of these features, save for the compile part at least.

Maybe “the bubble” had something to do with it, too. I’m referring to that period when everyone thought they had the surefire way to make a buck off the Web, and when the public was just starting to get online and figure out how cool a place the Web was. There were 80-hour work weeks, powered by Jolt Cola, jelly donuts, and the incessant chant of some flower shirt–wearing, Segway-riding (okay, Segway wasn’t out then, but work with me here!) recent college grad with an MBA, who promised us that all those stock options would be worth more than we could count. Maybe that caused everyone to just slap the code together so it at least appeared to work, in a pointless attempt to implement the business plan, and is really what caused all the trouble.

Yeah, you’re right, probably not. Ahem.

The good habits that developers had learned over time, such as code formatting, commenting, and logical code structure, had to essentially be relearned in the context of JavaScript. And, of course, those who hadn't done much programming before had to learn it all anew. But learn they did, and from that point, JavaScript started to become something
“professional” developers didn’t thumb their noses at as a reflex act. It could start to become a first-class citizen, now that developers had the knowledge of how to do it right.

Of course, the last step was yet to come.

What’s Old Is New Again: JavaScript in the Present

We’ve arrived at the present time, meaning the past three to four years. JavaScript has really come into its own.

The whole Ajax movement has certainly been the biggest catalyst for getting JavaScript on a more solid footing, but even a bit before then, things were starting to come around. The desire to build fancier, more-reactive, user-friendly, and ultimately fat-client-like web applications drove the need and desire to do more on the client. Performance considerations certainly played a role, too, but I suspect a lot smaller one than many people tend to think.

The bottom line is that JavaScript has moved pretty quickly into the realm of first-class citizen, the realm of “professional” development, as Figure 1-2 demonstrates. Perhaps the best evidence of this is that you can now find terms such as JavaScript Engineer, JavaScript Lead, and Senior JavaScript Developer used to describe job offerings on most job search sites. And people now say them with a straight face during an interview!

Figure 1-2. No longer the object of scorn, JavaScript now gets the respect it deserves.

So, aside from Ajax, what are the reasons for this relatively current trend toward respectability that JavaScript seems to have earned? Let’s have a look.

Browser compatibility isn’t so much of an issue anymore, at least when it comes to JavaScript itself (DOM differences are still prevalent, though). Object-oriented design has found its way into JavaScript, leading to much better overall code structure. Graceful degradation is commonplace nowadays, so web sites continue to work even when JavaScript isn’t available. Even internationalization and accessibility for those with disabilities is often factored into the design of the code. In fact, this is all a way of saying that JavaScript developers have learned enough that they can be far more responsible with what they design than they
ever could before. JavaScript is no longer something for script-kiddies to mess with but now provides a respectable vocation for a developer to be in!

The White Knight Arrives: JavaScript Libraries

JavaScript libraries have grown leaps and bounds over just the past few years, and this is perhaps one of the biggest reasons JavaScript has gone from pariah to accepted tool in the toolbox of good web developers. It used to be that you could spend a few hours scouring the Web looking for a particular piece of code, and you would eventually find it. Often, you might have to, ahem, “appropriate” it from some web site. Many times, you could find it on one of a handful of “script sites” that were there expressly to supply developers with JavaScript snippets for their own use. If you were fortunate enough to find a true library rather than some stand-alone snippets of code, you couldn’t count on its quality or support down the road.

Using libraries is a Good Thing™, as Ogar from Figure 1-3 can attest (and which Tor is finding out the hard way).

Larger libraries that provide all sorts of bells and whistles, as exist in the big-brother world of Java, C++, PHP, and other languages, are a more recent development in the world of JavaScript. In many ways, we are now in a golden age, and you can find almost more options than you would want! While some libraries out there focus on one area or another (graphical user interface, or GUI; widgets; Ajax; effects; and so on), other libraries try to be the proverbial jack-of-all-trades, covering a wide variety of areas such as client-side storage, widgets, Ajax, collections, basic JavaScript enhancements, and security.

![Figure 1-3. Be like Ogar, not Tor: use those libraries; don’t always reinvent the wheel!]

The one thing all these libraries have in common is that their quality is light-years beyond what came before, and they all will make your life considerably easier! Not only are they more solid, but they are better supported than ever before with real organizations, both commercial and otherwise, backing them up.

There’s usually no sense in reinventing the wheel, so libraries are indeed a good thing. If you are implementing an application using Ajax techniques, unless you need absolute control over every detail, I can’t think of a good reason not to use a library for it. If you know that your UI design requires some more-advanced widgets that the browser doesn’t natively provide, these libraries can be invaluable.

There are oodles of libraries out there today, all with their own pluses and minuses. Most of the bigger names out there—jQuery, the Yahoo! UI Library (YUI), Direct Web Remoting
(DWR), Prototype, script.aculo.us, MooTools, and Rico, just to name a few—are all top-notch, and you can hardly go wrong picking one or the other. However, they all have a somewhat narrow focus, for the most part. jQuery is about accessing parts of the DOM as efficiently as possible. YUI is about widgets. DWR is about Ajax. Script.aculo.us is about effects, as is Rico. Prototype and MooTools are both a bit more general-purpose in nature, which is closer in philosophy to the library we’re here to discuss, that of course being Dojo.

Without further ado, let’s meet the star of the show, the Dojo Toolkit!

The Cream of the Crop: Introducing Dojo!

Dojo, or more precisely, the Dojo Toolkit (but from here on out it’ll be just Dojo for short) is an open source JavaScript library. No surprise there! Dojo is licensed under either the Academic Free License (AFL) or the BSD license. The AFL is the more liberal of the two and is favored by the Dojo Foundation, which is the parent foundation under which Dojo lives. However, the Free Software Foundation (FSF) has created some ambiguity between the AFL and the two licenses that the FSF is famous for concocting: the GNU General Public License (GPL) and Lesser General Public License (LGPL). So, the fine folks at Dojo give you the choice to use the BSD license or the AFL, the BSD license being compatible with both the GPL and LGPL licenses, as the FSF says.

Dojo seeks to be your “one-stop shop” for all your JavaScript needs. Many libraries out there focus on just one or two areas, which means that you as a developer will usually have to combine two or more libraries to get everything you need done. With Dojo, that’s not the case. Dojo covers all the bases, from core JavaScript enhancements to UI widgets, from browser-independent drawing functions to offline storage capabilities.

The kicker to it all, though, is that Dojo does this well! I bet you’ve seen libraries that purport to do it all, and in fact they do pretty much cover all the bases, yet they fail in one important regard: they’re not very good! They tend to have lots of bugs, incomplete functionality, and no support for any of it. Dojo turns that idea upside-down and not only gives you virtually everything you’d need to create rich modern web applications, but enables you to get everything from the same source, and that’s definitely a plus.

There’s an underlying philosophy to Dojo, and a big part of it is what I just talked about. But before I get any further into that, let’s step back and take a quick look at how Dojo came to be in the first place.

Let’s Take It from the Top: A Brief History of Dojo

The year was 2004. The air was musky. Oh wait, sorry, I thought I was writing a novel for a moment. Anyway, the year was 2004, early 2004 to be precise. A gentleman by the name of Alex Russell was working for a company named Informatica doing DHTML development. Alex was the creator of a library called netWindows that allowed for creation of a windowed interface (much like the Windows operating system, which was unfortunate as you’ll see!). He began looking to hire someone to help him in that work, and so some prominent members of the DHTML programming community on the Web were contacted. This need led to a more generalized discussion among the DHTML community about the future of DHTML, and web development overall.

Ultimately, the job (one of them as it turns out) went to Dylan Schiemann (the other job went to David Schontzler, who worked at Informatica for the summer). As the job progressed,
Alex and Dylan, along with some others, started to have discussions about developing what would be a “standard” library for JavaScript, much like the standard libraries that exist for most other languages, such as C for instance.

A new mailing list (ng-html, which later became the dojo-developer list) was created, initially with folks including Aaron Boodman, Dylan Schiemann, Tom Trenka, Simon Willison, Joyce Park, Mark Anderson, and Leonard Lin on it. Discussions on licensing, intellectual property (IP) rights, coding standards, and such began in earnest. After not too long a time, the really difficult job began.

That job, of course, being what the heck to name the whole thing!

A while earlier, Alex had received a cease-and-desist letter from Microsoft over his use of the word windows in his netWindows project. Seriously! I mean, I like to think I’m a pretty creative guy, but even I couldn’t make up something that wacky! Which reminds me, I need to go trademark the terms sock, sky, person, and water. But I digress. The folks on the mailing list tossed around a bunch of ideas with one of the stated goals (I’d be willing to guess one of the top one or two goals!) to make it something that wouldn’t get them sued. Leonard Lin proposed the name Dojo, and the rest was history.

The first early code was committed by Alex and Dylan, with the support of Informatica. Soon after, two companies, JotSpot and Renkoo, got behind the project and provided early support. By this point, good momentum was starting to build on the mailing list, with everyone starting to contribute bits and pieces they had lying around, and a JavaScript library was born!

The Dojo Foundation was founded in 2005 to house the copyrights for the Dojo code. The foundation is a 501(c)(6) organization, which is a type of tax-exempt nonprofit organization under the United States Internal Revenue Service code. Alex serves as the president of the foundation, and Dylan is the Secretary/Treasurer.

None of this is terribly important to us code monkeys, I know! But it’s interesting to know the history nonetheless, and with those details embedded in our brains, we can move on to more-interesting things, starting with the underlying philosophy behind Dojo.

---

**CODE MONKEY? WHAT DID YOU JUST CALL ME?!?**

Some people consider the term code monkey to be derogatory, but I never have. To me, a code monkey is someone who programs for a living and enjoys writing code. I suppose by that definition you wouldn’t even have to earn a living from programming, so long as you love hacking bits. Either way, it’s all about the code!

By the way, just because someone is a code monkey doesn’t mean they can’t do architecture, and vice versa. If you like all of the facets of building software, if you like being up ’til all hours of the night trying to figure out why your custom-built double-linked list is corrupting elements when you modify them, if you like playing with that new open source library for no other reason than you’re curious, if you like the feeling you get from seeing a working application come spewing out the back end of a development cycle (even if it’s some otherwise dull business application), then you’re a code monkey, plain and simple, and you should never take offense to being called that name.

Of course, some people do mean it in a derogatory way, and you’ll know who they are, in which case you should be a good little code monkey and throw feces at them. (Frank Zammetti and Apress cannot be held liable if you actually follow this advice!)