sitepoint®

BUILD MOBILE
WEBSITES AND APPS FOR SMART DEVICES

BY EARLE CASTLEDINE
MYLES EFTOS
& MAX WHEELER

WHIP UP TASTY MORSELS FOR A NEW GENERATION OF MOBILE DEVICES

Tap into the amazing possibilities of mobile web development!

Welcome to the sample chapters of *Build Mobile Websites and Apps for Smart Devices*.

No doubt you've gathered that this book is all about developing and designing for mobile devices. The full version will show you how to turn a website into something much more amazing.

Your mobile journey will take you from basic website to sexy mobile site, from cool mobile app to lucrative and seductive native app.

Better services and smaller, cheaper devices have brought about a huge explosion in mobile technology, far outpacing the growth of any other computing cycle.

If you need convincing as to the mobile web's impact, simply look around you. Everywhere you go, people are accessing the Web from their devices. Check out these statistics:

- By the year 2013, consumers will be buying more smartphones than PCs and Laptops.[1]
- Since the launch of the iPhone, more than four billion apps have been downloaded, with an average of 47 apps per user. Android and iPad app stats are also in the millions.[1]
- Worldwide mobile browsing has increased 148% in just a year.
- The number of users accessing Facebook and Twitter through their mobile devices has more than doubled in a year.[2][3]

Clearly, the need to develop for mobile devices is very much alive, and will only become more necessary as time goes on.

This book will ensure you're learning the skills needed in order to capitalize on this opportunity. Plus, the information is presented in a fun and fresh style, so that it's easy for you to make the most of this new technology right now.

Enjoy!

---

[1] Internet Trends - Presentation from CM Summit, Morgan Stanley, June 2010

[2] http://www.slideshare.net/kleinerperkins/kpcb-top-10-mobile-trends-feb-2011

[3] http://www.facebook.com/press/info.php?statistics and http://gs.statcounter.com/#mobile_vs_desktop-ww-monthly-201005-201105

What's in This Excerpt

Preface

Chapter 1: Introduction to Mobile Web Design

We'll start by covering what designing for mobile devices means. You'll be guided through the process of designing and building a mobile web application, including what needs to be considered when producing for a mobile context. Although we'll focus primarily on designing for smartphones, much of the advice provided can apply to any form of mobile device.

Chapter 2: Design for Mobile

Naturally, we want to deliver the best content and experience to our users, but what's key for mobile applications is the *context* in which users will access that information. In this chapter, we'll address how this influences our role as web developers and designers, and what changes we need to make.

Chapter 4: Mobile Web Apps

This is where we make our mobile website more interactive by turning it into an application to sell in the app marketplaces. We'll recreate native behaviors in a web setting, being mindful of our limitations whilst playing up to our strengths—transforming our websites into apps that are fun to use.

What's in the Rest of the Book

Chapter 3: Markup for Mobile

In this chapter, we'll focus on the HTML5 and CSS3 features we'll employ to create mobile web apps using standards-based web development techniques. A page with well-structured HTML and clean markup will display and be usable on any device, be it desktop or mobile.

Chapter 5: Using Device Features from Web Apps

The rise of the smartphone has brought with it all sorts of advanced features—the functionality of which you'd expect could only be served by the native app. Luckily for us, the speedy implementation by mobile browsers of emerging standards has meant that web apps are gaining ground in functionality. This chapter will explore how we can make the most of event-based APIs interacting with the new hardware.

Chapter 6: Polishing Up Our App

Now that we've done the groundwork, it's time to apply some spit and polish to our app. In this chapter, we'll explore what's available to help us manage inconsistencies between web and native applications, in order to refine and produce a scintillating app for the marketplace.

Chapter 7: Introducing PhoneGap

In this chapter, we'll address how to convert our web app into a native app that can run on several platforms with the help of the PhoneGap framework. We'll look at installing all the required software to develop for iOS, Android, BlackBerry, and webOS, as well as PhoneGap itself.

Chapter 8: Making Our Application Native

In the final chapter, we unleash our web app into the native environment. We'll cover what's involved in customizing our app for each of the main platforms, as well as some necessary tweaks to iron out any inefficiencies that might stop us from gaining marketplace approval. Finally, we'll look at simulators as we detail the all-important testing process.

Appendix A: Running a Server for Testing

Testing sites on mobile devices can be a little trickier than testing on desktop browsers. In this short appendix, we'll look at a few simple web servers you can use to deliver pages to your phone from your development machine.

Chapter 1

Introduction to Mobile Web Design

Are you ready to step into the next big arena of web design? *Build Mobile Websites and Apps for Smart Devices*, as the name suggests, is all about designing for mobile devices. It's about designing for the future. This book will guide you through the process of designing and building a mobile web application from scratch. We'll take a look at what you should consider when designing in a mobile context—building the base of our application using web standards, and layering interaction on top of that base. Ultimately, we'll have our application up and running in a native wrapper so that it can be downloaded from the various app marketplaces. This book will focus on building for phone-sized devices, though many of the concepts and techniques can be applied to other mobile devices and contexts, such as tablets or netbooks.

From a technical perspective, we're going to be talking about the same technologies we're used to building with; HTML, CSS, and JavaScript are going to form the basis for (almost) everything we cover. So you will need a basic understanding of those technologies at the very least.

What does it mean?

First of all, let us make sure we are on the same page. You may well ask, "What do you mean by *mobile*?" The answer is: many things. On the surface, building for the mobile web may appear to be not all that different from building for any other web application or site; we're simply optimizing for viewing on mobile devices. Dig a little deeper, though, and there's a lot more we need to think about.

Discussions about the mobile web tend to focus on the devices and their capabilities—things like the latest iPhone, the newest Android phone, or this week in webOS. It's a rapidly changing landscape and thus an exciting time for web development, so it's easy to get caught up in discussions of the technical requirements and solutions for targeting mobile devices. But this misses the great opportunity we have with mobile design, because, ultimately, it's about people, not devices. The definition Barbara Ballard gives in her book, *Designing the Mobile User Experience*, is right on the money:[1]

> Fundamentally, "mobile" refers to the user, and not the device or the application.

People, not things. Mobility is more than just freedom from the confines of our desks. It's a different context, a distinct user experience. Strangely enough, people use mobile apps *when they're mobile*, and it's this anywhere-and-everywhere convenience of mobile design that makes mobile applications incredibly useful, yet so hard to design. We need to think hard about who we're targeting and what they want or require. Our focus has to be on having our application shine in that context. And while, for much of this book, we'll be focusing on the technical implementation, we'll be keeping Ballard's definition at the forefront of our decision-making.

Why does it matter?

Estimates put the combined number of smartphones and other browser-equipped phones at around 1.82 billion by 2013, compared to 1.78 billion PCs.[2] Reliable stats on mobile browser usage are notoriously difficult to find, but regardless of the source, the trend is clear. According to StatCounter, the mobile share of overall web browsing is currently sitting at 4.36%, and while that figure may seem small, bear in mind that's a whopping 430% increase over the last two years. And this is just the very beginning of mobile browsing. We're never going to spend less time on our phones and other mobile devices than we do now. Inevitably, more powerful mobile devices and ubiquitous internet access will become the norm. And the context in which those devices are used will change rapidly. The likelihood of our potential customers being on mobile devices is higher and higher. We ignore the mobile web at our peril.

The Natives Are Restless

The inevitable decision when designing for the mobile space is the choice between building a native application or a web application. Let's first define both of those terms. A **web application** is one that's accessed on the Web via the device's browser—a website that offers app-like functionality, in other words. A so-called **native application** is built specifically for a given platform—Android or iOS, for example—and is installed on the device much like a desktop application. These are

[1] Hoboken: Wiley, 2007

[2] http://www.gartner.com/it/page.jsp?id=1278413

generally made available to consumers via a platform-specific app marketplace. Most famous among these is Apple's App Store for the iPhone and iPad.

Let's now take a look at the pros and cons of native apps and web apps. As a general rule, native apps offer a superior experience when compared to web applications; the difference is even more pronounced on slower devices. Native applications are built, optimized, and, most importantly, compiled specifically for the device and platform they're running on. On iOS, this means they're written in Objective-C, and on Android, in Java. In contrast, web applications are interpreted; that is, they have to be read and understood on the fly by the browser's rendering and JavaScript engines. For iOS, Android, BlackBerry, Symbian, and webOS, the browser engine of choice is the open source WebKit project—the same engine that powers Safari and Chrome. For Windows Phone 7, the engine is currently a version of Internet Explorer 7, though Microsoft have announced plans to change that to the rendering engine inside Internet Explorer 9. This extra layer between our code and the device means that web applications will never perform as well as native apps, and that's problematic if we're building an app that requires high-resolution 3D graphics or a lot of number crunching. However, if we're building something simpler, a web app will do the job just fine. There will still be a difference in performance, but we will be able to provide a good user experience nonetheless.

The need for web applications to be interpreted by an engine also means we're bound to that engine's limitations. Where native applications can access the full stack of methods exposed by the operating system, web applications can only talk to the operating system through the browser engine. This means we're limited to the APIs that are made available by the browser. In iOS, for example, native applications have access to a whole set of functionality that's unavailable through Mobile Safari; for example, push notifications, the camera, the microphone, and the user's contacts. This means we could never build a web application that allowed users to upload photos to a service like Flickr or Facebook. It's simply not possible. That said, there are a range of device functions that are exposed through the browser: the Geolocation API lets us find out where our users are (if they permit us); the DeviceOrientation API gives us access to the gyroscope and accelerometer data; and with the Web Storage API we can save data between browsing sessions. Throw in HTML5 audio and video, gestures through browser touch events, CSS transitions and transforms, and 3D graphics using WebGL, and we can see that the gulf in functionality is narrowing. But it's likely that there'll always be something—usually the latest and greatest feature—that we're unable to use.

So, if we agree that native apps are the ideal, why are we talking about building web apps at all?

The Problem with Going Native

One issue with building a native application is market fragmentation. Because they are platform-specific, it begs the question: what platforms do we target? Should our application be in Apple's App Store first, or the Android Marketplace? What about BlackBerry or Windows Phone 7? Keep in mind that for each platform we want to support, our app will have to be rewritten. In an ideal

world, we'd build a native application for all those platforms and more, but in the real world, our resources are limited; so we're forced to choose which platforms—or more accurately, which users—will miss out. Building a web application, however, means that as long as those devices come fitted with a web browser, we can build a single application from a single codebase that services all those platforms and more. The issue of fragmentation applies to browsers, hence web applications as well, but this is a familiar problem to web designers, and the differences are usually minor.

Another issue is the speed of development. As web professionals, we have a wealth of experience in building and managing web applications. Learning a whole new set of development tools, or hiring a person with those skills, takes time, effort, and money. We need a reason to justify that hassle and expense, rather than just simply betting on the skills we already have. The counter argument is that such reasons are predicated on what is best for our business, not what is best for our users, and that's a fair point. It's a delicate balancing act. We're trading user experience for familiarity, development speed, and platform flexibility. Of course, we want to make the best possible application for our users whatever their preferred platform, but an app that gets built offers a far greater user experience than one that never sees the light of day.

In recent times, some high profile companies have weighed up this equation and then put their efforts behind the Web. 37signals, purveyor of various web-based productivity applications, including Basecamp and Highrise, eschewed the native app path in releasing Basecamp mobile:

> Eventually we came to the conclusion that we should stick with what we're good at: web apps. We know the technologies well, we have a great development environment and workflow, we can control the release cycle, and everyone at 37signals can do the work.
>
> […] we work in HTML/CSS/JS every day and have been for years. Gains we make on the desktop can make it into mobile, and gains we make in mobile can make it back to the desktop. It's the right circle for us.[3]

For the team at 37signals, dedicating money and resources was not the issue. They simply decided that building a web application provides a better experience for more users, and that building it using technologies they're familiar with gives them better control over the application in its entirety. Netflix came to a similar conclusion. Its application for the PlayStation 3 is written entirely in web technologies, enabling its developers to test and iterate the application continuously so that the best result is achieved for users:

> Our core mandate is to relentlessly experiment with the technologies, features and experiences we deliver to our members. We test every new idea, so we can measure the impact we're having on our customers. […]

[3] Jason Fried on *Signal vs. Noise*, 1st February, 2001 [http://37signals.com/svn/posts/2761-launch-basecamp-mobile]

> That's where HTML5 comes in. The technology is delivered from Netflix servers every time you launch our application. This means we can constantly update, test, and improve the experience we offer. We've already run several experiments on the PS3, for example, and we're working hard on more as I write this. Our customers don't have to go through a manual process to install new software every time we make a change, it "just happens."[4]

Even Facebook, a company with more than a modicum of engineering resources (having produced the number one iPhone app of *all time*), finds it difficult to manage the platform fragmentation and is embracing web standards as the future of their mobile strategy.[5]

Mobile web apps offer several advantages over native apps, and though they face some design, development, and deployment challenges, they're a powerful cross-platform solution that's both scalable and affordable.

APIs enable

Despite 37signals's decision to stay away from native app development internally, there are no less than ten native clients for its Basecamp web application currently for sale in Apple's App Store. The comprehensive API it makes available means that third-party developers have been able to build native applications on top of Basecamp, while still allowing 37signals to control the level of interaction allowed with users' data. A well-constructed API means that your users can build your apps for you, some that you might not have expected.

Start at the Beginning

"We *need* an iPhone app." Yes, you might, but a native application for the various platforms isn't the be-all and end-all. There has to be more behind our decision than "but everyone else has one." We need to consider whether building a mobile application—whatever the technology—is the right decision for us and our users. If you have a coffee chain with 1,000 locations nationwide, perhaps a mobile application that helps your customers find your nearest store is a great idea. But if you represent the neighborhood's hipster bicycle-shop-*cum*-café-bar, perhaps a simpler alternative is more fitting.

Do people need what we're offering? Why would people want to use our application while they're mobile? Where will they use it? What are the outcomes for us as a business?

A great way to get answers to those questions is to examine information that's readily available to you. Look at your current website statistics: how many visitors are viewing it on their mobiles?

[4] John Ciancutti, *The Netflix "Tech" Blog*, 3rd December, 2010 [http://techblog.netflix.com/2010/12/why-we-choose-html5-for-user.html]

[5] http://networkeffect.allthingsd.com/20110125/facebook-sets-mobile-sights-on-html5/

What devices are they using? Which content are they looking at? Such data can provide an insight into what people are seeking in a mobile context. Of course, the data will be influenced by the constraints of your current website, so any conclusions you glean should only form part of your decision process.

What if you have no data to be mined? Well, you could always try talking to your users; there's no harm in asking people what they want. In simple terms, it's probably whatever you're selling, as quickly as possible. If you're a florist, they want flowers—*now*. Own a café? They want coffee, *now*. Whatever your product or service, if you can create an application that meets those demands, it will be tremendously gratifying for your users (and will make you money).

The App Store Effect

The success of Apple's App Store can't be ignored: there's an undeniable marketing advantage to having an app that appears in such a popular forum, and having your icon in the middle of a user's home screen gives your app exposure in a way that a bookmark does not. In addition, the path to monetization is very clear: the various application marketplaces bring customers, and those customers bring money. We're going to build a mobile web application, but that doesn't mean we have to miss out on a potentially lucrative outlet for our product. This is where a web-native hybrid app can come in. But we're getting ahead of ourselves—all this and more will be covered in Chapter 7.

An App is not Enough

The biggest argument for making a mobile application using web technologies is that we're going to have to do at least some of that work anyway. Users, rightfully, will expect the website we already have to work on their mobile devices. No assumptions should be made about a user's device or its capabilities—an underlying principle of the Web at large—because inevitably those assumptions will be wrong. A native app is not the solution to that problem.

We've identified that mobile design is about context, but it's also about speed. We're aiming to give our users what they want, as fast as possible. Having a fantastic native application is good only if users already have it installed. Asking our users to go to an app marketplace to download a separate application—usually a large, upfront hit—can be too much to expect if they're already on the go and relying on mobile internet coverage. Providing a version of our site to mobile users is going to be important regardless of whether or not we have a native application. So what do we do?

Option One: Nada

Doing nothing is seriously one option, and shouldn't be dismissed. The new breed of smartphones make it incredibly easy to navigate around a large and complex page. The home page from *The New York Times*, for example, contains an enormous amount of information across a range of topics. If we take a look under the hood, though, we can see that this volume of information comes at a price;

to download the front page of http://nytimes.com/ takes up almost 1MB of data, as Figure 1.1 reveals using Chrome's Web Inspector tool.

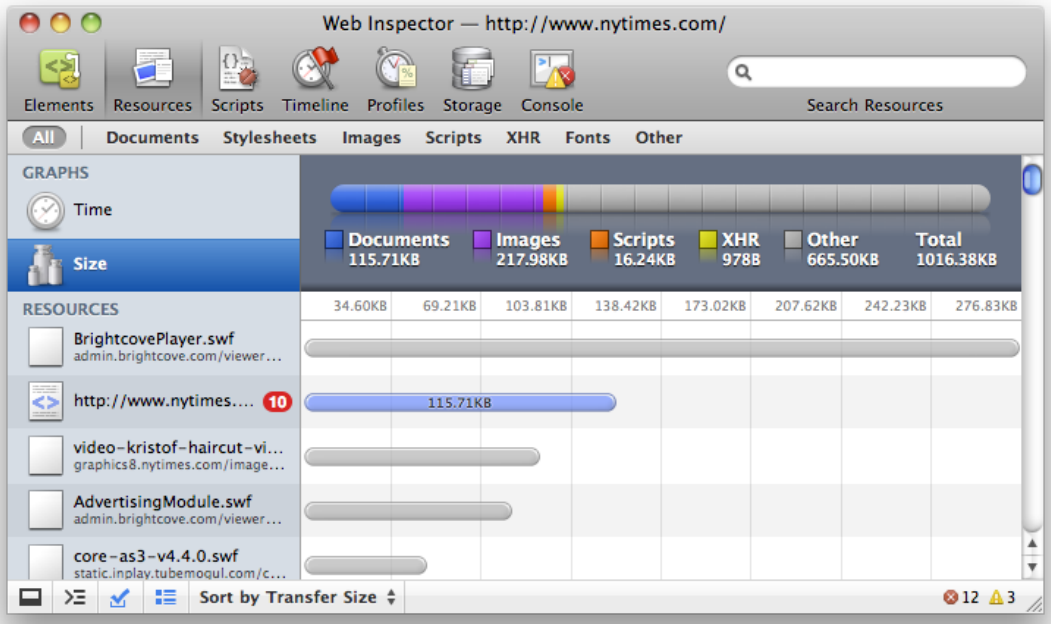

Figure 1.1. Chrome's Web Inspector reveals the true cost of full-featured pages

Sure, 3G coverage is fairly decent these days, but we're still asking our users to wait a good four to five seconds before they can interact with our site. This isn't terribly mobile- or user-friendly. If we do choose the path of least resistance, it's imperative that we build our site using well-structured and meaningful markup. Adhering to all the conventional best-practices for desktop web design will bear even greater fruit when it comes to mobile. Lightweight, CSS-based layouts, content-out design, and a focus on accessibility and usability matter even more when screen size, attention, and bandwidth are limited.

Option Two: Transform and Move Out

Responsive design to the rescue! Well, sort of. If you somehow missed the seminal article from Ethan Marcotte on this topic, we'd strongly recommended that you take the time to read it.[6] The phrase **responsive web design** refers to the art of using CSS media queries, fluid grid layouts, and fluid images to respond to the resolution of the user's device (or browser window) and adapting your design to provide the best layout for any given resolution.

[6] http://www.alistapart.com/articles/responsive-web-design/

It's a beautifully simple technique—if a little mind-bending at times—and worth looking into. Media queries are an extension of the familiar media-type attributes we've long used to deliver print stylesheets to our HTML pages. The difference is that instead of only allowing a singular context for those CSS rules, we can query (hence the name) the physical characteristics of our user's device. This means we can deliver different stylesheets (or bits of stylesheets) to only the devices that fit the criteria we specify. In the example below, we're saying, "Only load the **mobile.css** file if the viewport is at most 480px wide":

```
<link rel="stylesheet" type="text/css" media="screen and (max-width: 480px)"
➥href="mobile.css" />
```

Of course, we're not limited to inspecting the width of the device in question. Among the many supported features in the media queries spec, we can ask about:

- width and height (as in the above example)
- screen resolution
- orientation
- aspect ratio

The real power, though, is the ability to combine these queries into more complex rules. Want to serve some styles to a high-resolution device in landscape orientation? No problem:

```
<link rel="stylesheet" type="text/css" media="screen and
➥(min-resolution: 300dpi) and (orientation: landscape)"
➥href="mobile.css" />
```

That's fine and dandy, right? We can use this approach to serve a great mobile site, a great desktop site, and everything in between, from the same codebase.

Alas, that's not quite the case. The responsive design approach has limitations. In a sense it's a bit like putting lipstick on a pig. Reformatting the layout can make our content more palatable on a mobile device, but it's still only window dressing. There's an argument to be made that responsive design is actually worse than doing nothing, because you're forcing your users to download resources they may never be able to see. Fortunately, some of these problems can be mitigated with some careful planning.

First and foremost: *build for mobile first*. Whereas the the inclination is to use media queries as a means to add a bit of extra love to a near-complete desktop site, what we should be doing is the exact opposite. Start from a base of simplicity and layer the complexity on top for a more powerful desktop experience:

```
<link rel="stylesheet"
➥ media="screen and (min-width: 939px)" href="desktop.css" />
```

It's a concept we're quite familiar with: progressive enhancement. There's no way for us to avoid sending the same HTML to each device; however, if we're careful about how we construct our pages and serve our stylesheets, we can ensure an optimal experience for mobile users—providing the content they're most likely to want up front. We're minimizing the overhead without atrophying the desktop experience.

Option Three: Forever Alone

A more common option is to build a completely separate website for mobile users. Such a site can usually be found on an m. or mobile. subdomain (or both). Choosing this method means we can craft an experience that's focused solely on the needs of our users when they're out and about. Perfect.

Well, almost. There are some downsides to this approach. Having a separate mobile site usually means relying on some form of user agent detection to decide which device receives which edition of our site. For example, "serve the mobile site to users on iPhones, serve the desktop version to Firefox users" and so on. While great in theory, this sort of user agent sniffing (detecting the user's browser based on the information it provides about itself in requests to our server) is notoriously unreliable; the user agent string can be easily changed in many browsers (and often is to specifically get around this technique). Even if we were to make our mobile. site correctly detect the current crop of mobile browsers, we can cause unintended problems for users of devices we don't know about yet. It's a question of choice: do we want to force our users down the route we've chosen?

The solution is dead simple: allow them to choose their own path by providing a link to our standard site on the mobile version (and vice versa). Then, respect that decision. There's nothing wrong with encouraging our users to start at our mobile site, but we shouldn't restrict them from accessing everything they could normally access.

Facebook is a good example of the right behavior, addressing the reasons you may want to allow your users to switch between the standard site and the mobile site. It offers two mobile sites: touch.facebook.com to cater for mobile users on touch-enabled smartphones, and m.facebook.com for users of non-touch devices. Both sites let you perform all the normal tasks and interactions that you'd expect Facebook to offer: you can read and respond to messages, post status updates, and view the wall of activity that's at the heart of the site. Still, we can't do *everything* that the standard desktop site lets us do—upload photographs or edit our profile, for example. If we absolutely have to perform a task that's only possible on the standard site (or works better on the standard site), both of Facebook's mobile editions provide a handy link in their footer to switch versions. The key is to always allow your users easy access to the full-featured site—don't wall them off from any functionality. Separate, don't segregate.

A Note on Frameworks

When doing research into building web applications for mobile devices, you'll no doubt come across projects that purport to provide cross-platform development frameworks for mobile. The most prominent of these are Sencha Touch[7] and the jQuery Mobile[8] projects. We'll skip delving into the specifics of their implementation, but they're worth taking a quick look at in order to decide whether or not they suit our purposes.

Both these frameworks are essentially JavaScript frameworks. In the case of Sencha Touch, the applications built with it are entirely reliant on our users' devices having a good JavaScript engine. We already know this is not always the case. View an application built in Sencha Touch without JavaScript and we get an empty shell that doesn't do anything. JQuery Mobile, meanwhile, chooses the more user-friendly approach of progressive enhancement; its applications are built in plain HTML, and their app-like behavior is layered on top. This is another trade-off—as you might be starting to grasp, the mobile world has a lot of those! Sencha Touch uses the all-JavaScript method because it means better performance—by virtue of building application logic in JavaScript rather than on top of the DOM. Regardless of their different approaches, the point to remember about these frameworks is that they both implement a whole host of features that replicate behavior and functionality present in native apps. So if you don't use all those features, you're including overhead that you have no need for.

This brings us to one of the design considerations when building a mobile web application: the **uncanny valley**. The uncanny valley is a theory that claims that the more lifelike a humanoid robot is, the more likely its appearance will cause revulsion in human beings. This idea can be applied to interfaces, so we should be aware of it when we start to look at the design (and behavior) of our application.

If it looks like a duck, quacks like a duck, and yet isn't a duck, how are our users supposed to treat it? By replicating the look and feel of a native application, mobile application frameworks set certain expectations—expectations that are, by definition, impossible to meet. The answer is simple: embrace the limitations. There's no need to pretend that we are outside the scope of the normal browser interface. Mobile isn't a curse; it's an opportunity to make an active decision about how we present ourselves. The mobile version of twitter.com doesn't attempt to behave like any of the native Twitter applications. It does what it's supposed to do: give you most of the information you want as quickly as possible.

[7] http://www.sencha.com/products/touch/
[8] http://jquerymobile.com/

Rolling Up Our Sleeves

All this discussion is wonderful, but isn't this supposed to be a book about, you know, *building* mobile web applications? Well, that it is; and while it's important to understand why we've chosen a particular strategy, it's much more fun to actually create something. So, what are we building? As luck would have it, we've been approached by a potential client to build a mobile version of their popular StarTrackr website. StarTrackr is a celebrity-spotting site that lets users log when and where they've seen a celebrity "out in the wild," and it's a perfect example of a task that's suited to the mobile web.[9] Let's review our options: we can do nothing (not sure our client will like that), craft a mobile-first responsive design, or create a separate (but related) mobile version of our site. It's a question of what we want—or rather what our client wants—to achieve. For StarTrackr, the client wants the user to be able to:

- see nearby spots (a location where a celebrity has been seen) and the various celebrity sightings at that spot
- find their favorite celebrities and see their recent sightings
- add a new sighting

If we look at that set of functionality, we can see we're talking about building a *web application*, not just a website. In simplistic terms, web applications are for performing tasks; websites are for consuming information. It's important to understand the distinction and how the techniques we've talked about are more appropriate to one or the other. We can do a lot with window dressing, but if our intention is to create a compelling and contextual experience for our users—and it is—we'll want to make the leap to a separate mobile application.

So let's get to it.

[9] Alas, StarTrackr is made up, so before you get too excited, you'll have to find an alternative means for your celebrity-spotting needs.

Chapter 2

Design for Mobile

Before we leap into designing our application, we'll look at some fundamental things to consider when crafting an interface for a mobile-centric app. It's easy to jump headfirst into creating the look and feel for an application. That's the fun part, right? The problem is that design doesn't start with Photoshop. First and foremost, it's about communication. Design is the process of organizing the information we want to present so that its structure is meaningful and instantly understandable. It's about controlling the navigation and flow of our application in a way that is clear, minimizes uncertainty, and feels efficient. As Jeffrey Zeldman, father of standards-based design, says in his seminal article "Style versus design":[1]

> Design communicates on every level. It tells you where you are, cues you to what you can do, and facilitates the doing.

We're looking to craft an interface that is functional, an interface our users can, well, *use*. We should be aiming to create an experience, or rather getting out of the way and letting our users create their own experience. The interface is simply the medium through which we enable it to happen.

For any website or web application we build, our goal should be to deliver the most appropriate content and experience to our users. What's crucial for for mobile applications is the *context*—the when and where—in which they'll be using that information. On our standard website, our users are quite likely sitting at a desk in front of a monitor with a keyboard and mouse at hand. Conversely, visitors who are browsing on a mobile device could be waiting in line, catching a train, lying on

[1] http://www.adobe.com/designcenter/dialogbox/stylevsdesign/

their couch, walking down the street, or perhaps even sitting on the toilet; and what's more, their screen is likely to be no larger than a few hundred pixels with a tiny (or onscreen) keyboard.

We need to think about how people use their mobile devices. More than the situations they're in, or the actions they're trying to achieve, consider how our users might physically be using their devices. How do they hold their phones? Are they using touch-enabled interfaces, or other input methods?

In general, we're going to adopt the same principles and thought processes that we'd normally apply to the Web—it's just that the issues are accentuated in the mobile space. The screen is smaller, input can be awkward, network connectivity is possibly slower and less reliable, and our users might be more distracted. We need to work harder than ever to maintain their attention and lead them to what they want (or what we want them) to do.

Build a Better Mouse

Many, if not most, of the new breed of mobile devices use touch as their main input method. While many of the principles we usually apply to interface design are the same, there are some shifts in mindset required.

Our fingers are remarkably dexterous, but they lack the same level of precision of a mouse. With some care, we can use a mouse to pinpoint a single pixel on an interface—but try touching a single pixel with your finger. The Apple Human Interface Guidelines for iOS specify a recommended hit target no smaller than 44×44px;[2] about 2,000 times bigger than our single pixel. Does this mean interfaces that rely on touch as their input method are a step backwards? Of course not. Touch as a mode of interaction is about removing the layer between us and our interfaces. What we lose in accuracy, we gain in having a better understanding of the interactions, because the experience is now tactile, and hence, more intuitive.

Interfaces that fail to accommodate the touch paradigm do so not because of any inherent failing of the input method, but because the designers have jammed too much onto a screen, or made buttons too small for fingers to easily tap. This is exacerbated by the inherent problem of touch as an input method: the very act of trying to touch an interface element obscures that element from our view. In addition, our users need to be able to understand and use our interface in a whole range of distracting contexts. There's a delicate balance between trying to fit as much information as possible into the smaller physical space of a mobile screen, and offering targets that are big enough for clumsy fingers to touch.

We should also never forget our users *without* touch screens! They're customers too, and their input method is likely to be extremely unfriendly. Older feature phones (and some smartphones) use

[2] http://developer.apple.com/library/ios/#documentation/userexperience/conceptual/mobilehig/UEBestPractices/UEBestPractices.html#//apple_ref/doc/uid/TP40006556-CH20-SW20

four-way navigation (often called a D-pad) as their primary input method, forcing users to scroll past several elements in our interface to reach the button they want. BlackBerry's browser uses a tiny trackball or scroll wheel to navigate the interface; hence, wherever possible, it's worth aiming to limit the number of elements on the screen.

Above all, it means simple interfaces. Simple means easy to understand, which leads to easy to use.

Simplicity is a feature, and while complexity is not necessarily a vice, we need to keep some perspective. As invested as we might be in the depths of functionality of our application, the behavior of our users is not likely to match our interest. Our users are likely to spend mere seconds or minutes in our application—if we can convince them to visit at all.

- What do they want?
- What do they expect?
- What do we want them to do?

We can't assume that users have the time (or the inclination) to figure out how our application works.

Hover Me

Designing for touch devices requires a shift in our thinking. Many of the techniques we've been used to employing no longer work, or at least don't quite work as we'd like. The most obvious of these is hover:

> Elements that rely only on `mousemove`, `mouseover`, `mouseout`, or the CSS pseudo-class `:hover` may not always behave as expected on a touch-screen device such as iPad or iPhone.
>
> —From Apple's "Preparing Your Web Content for iPad" guide[3]

Hovering as an interaction model permeates the Web. We're used to our mouse movements triggering changes to a page on hover: different colors or states for links, revealing drop-down navigation, and showing actionable items, to name a few. And as designers, we've readily embraced the possibilities that the hover state gives us. Most touch-based operating systems will do some work behind the scenes to try to ensure the interface deals with hover states in a non-catastrophic way, but we are going to have to start changing our habits. For example, in lieu of a hover state, consider:

- making buttons and hyperlinks obvious
- having content that doesn't rely on `:hover`; for example, increasing contrast on text
- avoiding drop-down menus without clear visual cues

[3] http://developer.apple.com/library/safari/#technotes/tn2010/tn2262/index.html

We might lose a little flair, but we'll gain clarity of interface.

Small Screens

There's no escaping that designing for mobile means designing for small screens. Mobile devices have displays that are significantly smaller—both in terms of physical size and resolution—than their desktop counterparts. This is an opportunity to be embraced.

Still, finding the right balance between information and interface to display on a small screen is a tricky problem. Too much detail and our interface will become cluttered and confusing. Too little and our users will have to work to find the information they need. This doesn't necessarily mean reducing content; it means reducing clutter.

In other words, don't be afraid of making an interface information-rich. A carefully designed interface can hold a wealth of information and communicate it effectively. Hiding information behind interaction may be the path of least resistance, but it's not necessarily the path we should take. People use their mobile devices to complete tasks or seek out information: they want to find out when the movie is playing, when the next train will arrive, or where the nearest café is. They would prefer not to spend hours exploring a sparse and delicately balanced interface. We want to give as much information as we can to our users without overwhelming them.

Cognitive Load

Simplifying the interface is really about reducing the cognitive burden we're placing on our users. This is essentially the overriding principle behind Fitts's Law.[4] For the uninitiated, Fitts's Law is a model of interaction that's become a fundamental when understanding user interface design. It states that the time to acquire a target—like say, moving a mouse over a button—is a function of the distance to and the size of the target. Simply put, the larger an item is and the closer it is to your cursor, the easier it is to click on.

A classic example of adapting to this principle is the menu bar in Mac OS X. It's a small target that's usually a fair distance from where your cursor normally sits; yet this is counterbalanced by placing the menu against the top of the screen, preventing users from overshooting the target. Being on the edge effectively gives the menu bar infinite height, resulting in fewer errors by the users, who reach their targets faster. For a touch screen, however, Fitt's Law has to be applied differently: our users aren't tied to the position of their mouse, so the origin of their movements is simply the default position of their fingers or thumbs. That position varies a lot on the device and its orientation; for example, with a mobile device, you might use the index finger of one hand or the thumbs of both hands.

Here's an example of this issue in a real application. Infinity Blade is an immensely popular game for iOS. It's available on both the iPhone and iPad, and uses the same interface for both devices.

[4] http://en.wikipedia.org/wiki/Fitts's_law

The game is played with the device in landscape mode, and the controls are anchored to the bottom of the screen (where your thumbs are), as you can see in Figure 2.1. On the iPhone, the "cast-spell" button is in the middle of the screen, within reach of either thumb. Yet this feature is less effective when found in the larger form of the iPad. The "cast-spell" button is still in the middle of the screen, but no longer within reach of our default hand position.

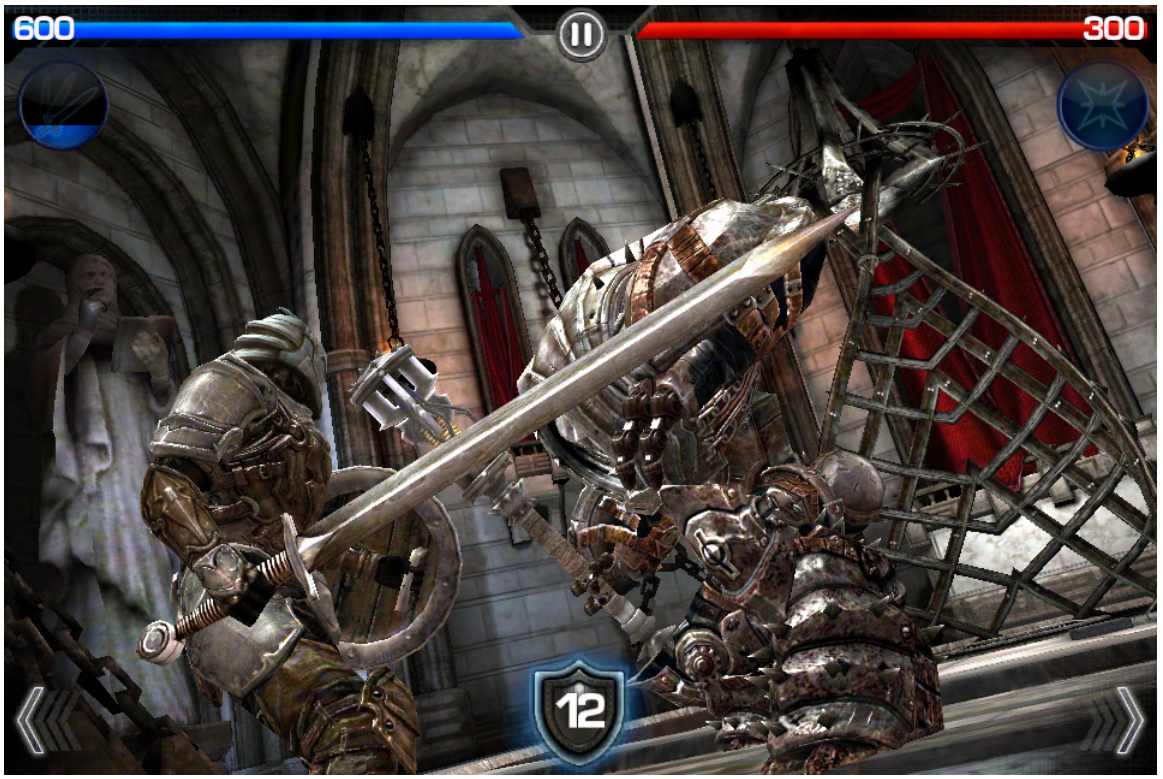

Figure 2.1. Infinity Blade on the iPhone

This is just one example, but it should serve to illustrate the importance of considering how your interface will be used on a real device. You need to think beyond "point-and-click"!

Standing on the Shoulders of Giants

A big part of the success of the iPhone, and the iOS ecosystem as a whole, has been Apple's focus on both the aesthetic and the experience of the platform and its applications. Apple did an enormous amount of work establishing the most common application design models while ensuring flexibility for their developers and consistency for their users. While our goal isn't to build an application that attempts to mimic the exact experience of using a native application, there's still plenty to be learned from examining the structure and design patterns used in mobile operating systems. Understanding the interfaces our users expect is important; it allows us to decide when it's worth trying to meet those expectations, and when to go in another direction.

Let's take a look at a few mobile design patterns we might find useful in our application.

The Carousel

Imagine some playing cards placed side by side in the screen, where users can flick between each "card" by sliding over the screen to the left or to the right.

The prototypical example of the carousel pattern on iOS is Apple's Weather app, seen in Figure 2.2. The Weather app assigns each city to a single card. One glance shows all the weather information we need for our city of choice without the distraction of what's happening elsewhere.

Figure 2.2. The carousel pattern in the flesh in Apple's Weather app

WebOS also uses the carousel pattern for switching between applications. Apps that use this pattern are normally information-rich but interaction-poor.

The carousel is the simplest pattern we'll look at—it usually consists of a single type of content organized in a linear set. What's nice about the carousel is that it's simple (which is good, remember?). The interface is minimal and the data structure is incredibly easy to understand. It also offers an implicit hierarchy of importance: the first items are the easiest to access, and are usually of the most interest to our users. The flip side to this structure is that there's no way to move between views more than one card away.

The Good

- It's simple to use.
- It utilizes a whole screen to show content.
- It requires a natural gesture for navigation.

The Bad

- It relies on gestures—the user has to swipe from card to card, which can be less intuitive than pressing buttons or menu items.
- All the information for a given page has to fit on the screen at the same time, otherwise the structure breaks down.
- Each page needs to be conceptually the same.
- Users have to progress through the sequence; they can't skip ahead.

Tab Bar

The tab bar pattern can be seen everywhere in iOS, Android, and webOS. For web designers and developers, tabs aren't exactly a new idea. We've been using them to establish hierarchy and group content into sections for many years. Conceptually, tabs in mobile applications are identical to those in desktop websites; the main difference is that the tab bar usually has a fixed position in mobile applications, and so always appears on the screen. It's interesting to note that on iOS, the tab bar component appears at the bottom of the page (near your thumbs), whereas on Android, the convention is to have the tab bar at the top of the screen (leading into the content).

The tab bar is useful for quickly establishing the structure of an application. It lets users move easily between the broad sections of an application, and also acts as an anchor point—the various selected states of the tab bar also signify where in an application the user currently is. As Figure 2.3 shows, the Twitter app for Android uses a tab bar to let users move between various modes of another user's profile.

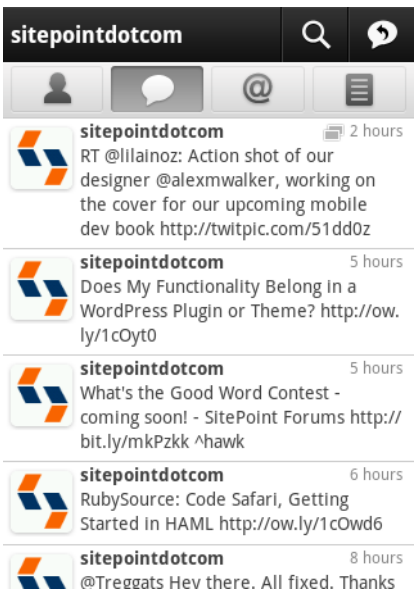

Figure 2.3. The tab bar in use in the Twitter app

Good

- It provides a familiar navigation for users.
- It allows easy switching between modes, views, or tasks.
- It indicates the current state/location of the app.

Bad

- Its hierarchy is flat—there's no easy way to have nested subpages.
- It always appears on the screen, taking up valuable real estate.
- It only handles up to five navigation items effectively, and is clunky beyond that.

Lists

Lists are the most commonly used design pattern for mobile applications. The list as an interface model is fairly self-explanatory: content is displayed in a vertical list, allowing users to scroll through the options. In iOS, they can be seen everywhere, featuring in all but the simplest of utility applications. While basic, they're also incredibly flexible. Lists can be used for presenting actionable options; acting as an index for a long list of content; and, most importantly, as a navigation hierarchy that lets users work their way down a tree structure.

It's as a navigation model that lists are most powerful. There are really no limits to the depths of navigational hierarchy that lists can accommodate and so, for applications with a structure more than one level deep, the list is almost universally turned to.

This pattern maps perfectly to the framework we're used to dealing with online. The list structure is a tree that can lead anywhere, and often it's used to let users drill down from an index of items to a detailed view of a single item. This is known as the **master/detail pattern**, a model that's used in desktop and mobile applications all the time. Just about every email application ever made uses this pattern of interaction, letting us quickly skim through the available items and then focus on a single one. We'll return to this idea a little later on.

For example, News.com.au uses the list pattern, allowing users to skim the headlines before moving into whichever story catches their interest, as you can see in Figure 2.4.

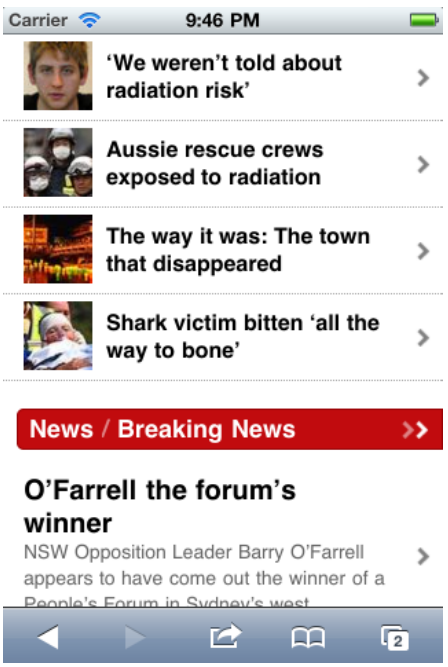

Figure 2.4. Lists are commonly used by news apps

The main limitation of lists is that as soon as a user moves down the tree, they lose the ability to move to any items on the levels above in one simple step. From four levels down, they would have to retrace back three levels to return to the top level—not ideal. To overcome this deficiency, the list structure is often combined with the tab bar pattern to create a strong, structured navigation with depth and flexibility.

Good

- It's flexible enough to handle lots of data.
- It's familiar and easy to understand.

Bad

- It's inherently hierarchical.
- Users need to return to the beginning to change paths.

Summary

Remember, these patterns offer a suggested structure—we don't *have* to use them. Familiarity and consistency can lend a design authority, but you can still break the mold. There are a myriad of examples of applications out there that eschew UI conventions to create delightful, intuitive, and unique interfaces for their users; by the same token, there are many apps that move away from these simple patterns without good reason and end up confusing and frustrating users.

We should always consider how breaking convention might enhance or detract from our app's primary task. If we're unable to design a better alternative to the conventional pattern, we probably shouldn't do it.

Putting It Into Practice

We've looked at some of the broader considerations when designing for mobile, so now let's address the business of our application. First, we need a plan. Our client has given us some high-level user scenarios, which we will need to examine in more detail to figure out what each means for our application. It's crucial to decide precisely which features we intend to deliver, and to whom. Only after we've figured that out can we ensure the look and feel of our app actually enables the task we'd like it to. So how do we go about that?

Thinking Big

In order to form the perfect feature set, we need to consider all the features our users *might* want. If we don't examine everything we might want to include, we're almost certainly going to miss some ideas that might make or break our app. Never mind if the list is long; it's better to start with grand possibilities and narrow our scope later. StarTrackr is a celebrity-spotting application, so our comprehensive feature list might include:

- Find sightings by location
- Find sightings by celebrity
- Sort celebrities by genre
- Search for a particular celebrity
- Find locations nearby that celebrities have been sighted at by address
- Find locations nearby that celebrities have been sighted at by GPS
- Favorite/follow a celebrity
- Favorite/follow a location
- Be notified when a particular celebrity is sighted
- Be notified when a particular location has a celebrity sighting
- View recent sightings
- Add a celebrity sighting
- Add a photograph of a celebrity sighting
- Add a video of a celebrity sighting

Putting Together a User Profile

As we talked about earlier, users are the most important consideration in the planning of our application. Who are they? What do we know about them? What are they doing? What do they want? What are the most important features to them? Why would they want to use our application? When are they going to use our app? What are they likely and unlikely to have an interest in? Such questions aren't always easy to answer, but if users are interested in the general concept for StarTrackr, we can probably make a few assumptions about them. We know they:

- like celebrities
- have an interest in celebrity gossip
- are more likely to be female than male
- are more likely to be younger than older (say, 14 to 25)
- are probably located near some celebrity hot spots, such as Hollywood or London

In the real world, our client would usually provide us with information on who their current users are and how they use the site. For our purposes, we'll make an educated guess: our typical user is a young woman who lives in a large city and enjoys celebrity gossip.

Deciding on a Core Feature Set

Start simple. Great mobile applications focus primarily on the tasks their users want to accomplish. In truth, we should be doing this with all our projects—mobile or not—but it's even more important with mobile web applications. It can be tempting to include more features in our application, thinking it will make it better. This isn't always so—in fact, it's hardly ever the case when talking about mobile design.

We have a list of features that we could include, and we have an idea of who our users are and what they're after. Next, we need to pare down our feature set to the most essential elements. This may seem easy enough, but deciding which features to leave out can be quite a challenge. Think about the minimum viable product: which features are essential for our application to be of any use at all. If we're brutal about whittling our list of functionality down to the basics, we can focus on three main features:

- Find sightings by location
- Find sightings by celebrity
- Add a celebrity sighting

And, secondary to that:

- Add a photograph of a celebrity sighting

For a first iteration at least, this gives us everything we need: celebrities are tied to locations, and we have the ability to filter through either of those data types—plus our users can add their own sightings while on the go. It's useful to look at this information and condense it into a single mission statement for our application; we can then return to it throughout the development to ensure we stay on track. For our app, this might be: "An app that lets people find and add sightings of their favorite celebrities."

Sketches

In any project, but particularly when we're designing for a form with which we're unfamiliar, sketching and **wireframing** is a great way to gain a sense of how the flow and overall structure of the app will work. What exactly is a wireframe, you ask? It's a low-fidelity visual representation of the layout of our application. We can use it to represent the basic page layout and navigational model of our app.

Since wireframes are considerably quicker to make than a fully fledged mockup, we can spend some time early on prototyping a range of approaches—testing our interface ideas and rejecting them promptly when they're not working. Once you start the design process, it's easy to get caught up in the small details: perfecting that shade of blue, or achieving just the right balance of type. At this early stage, sketches can provide a sense of the user experience as a whole, allowing us to spot potential usability problems before we commit too much time to building the interface.

How you build your wireframes is really a matter of personal preference. There are a whole host of tools out there that make the process very easy. Here are a few of the more popular options:

Pencil and paper

Sometimes the simplest tools are the best. There's something wonderful about the process of physically sketching a layout. It removes the distraction of the tools and lets us focus on the

task at hand. The downside to hand-drawn wireframes is that when we want to iterate our design, we'll have to redraw each layout.

Balsamiq, http://www.balsamiq.com/products/mockups
Balsamiq is a simple wireframing application with a range of common interface elements. It's an Adobe Air application that runs on Windows, Mac, and Linux, and has a hand-drawn look that's designed to keep you focused on the structure rather than the appearance of your mockups.

Mockingbird, https://gomockingbird.com/
An online tool, Mockingbird lets you build and share your wireframes in-browser. It has a similar sketchy style to Balsamiq.

Omnigraffle, http://www.omnigroup.com/products/omnigraffle/
Omnigraffle is a Mac-only application that, among its many features, is a fully featured wireframing tool. The real benefit of Omnigraffle is the range of interface stencils that are available for free. Check out Graffletopia[5] for all your wireframing stencil needs, including many mobile-specific templates for various platforms.

For our purposes, OmniGraffle will do the job just fine. Let's have a look at some sketches for StarTrackr. Because there are a bunch of stencils available out there for iOS wireframes, we're going to use the chrome from an iPhone as the frame for our sketches—but our interface will look the same on any platform.

First, we'll need to set up the correct dimensions and place some of the default elements in there. You can see our starting point in Figure 2.5.

That's our blank canvas. Now there are some standard interface elements we'll need to account for, as shown in Figure 2.6.

Figure 2.5. An empty wireframe containing only the device shell

[5] http://graffletopia.com

Figure 2.6. Adding the operating system and browser chrome

On iOS, the status bar weighs in at 20px, and in Mobile Safari, the toolbar at the bottom of the screen takes up 44px of vertical space. On Android, the browser chrome disappears from the screen once the page has loaded, but the status bar takes up 25px; the same is true of the browser on webOS.

Let's ignore these elements for a minute and look at the layout of our application in an empty screen. To start things off, it's often useful to establish the distinct objects that make up the application at large. For StarTrackr, these are easy to identify: we have celebrities (stars) and locations (spots) that are linked together by sightings. However we end up combining them in our interface, those three objects form the basis of everything we do. If we've identified the three main features we're going to focus on, what information or actions might each one contain?

Finding Sightings By Location

This one really is about the context. Our users are out and about, so they might be able to act on the information our app provides and go to the spot where a star has been sighted. While it might seem handy for a user to see *all* the most recent sightings, finding sightings that have happened *nearby* is probably much more useful. With that in mind, we'll want a list of locations—ordered by distance—that our users can drill down into.

A list, eh? Well, that sounds familiar. In fact, the list pattern is perfect for presenting this information. We want to let our users quickly flick through a series of locations, and then move into a detailed view of that spot if it piques their interest.

Figure 2.7 shows a wireframe of our list of spots, ordered by distance from the user. We're showing the important information on the left—the name and distance—and on the right, we've added the number of sightings at that given location. This will probably mean very little to our users initially, but there's no need to force content on them. It's okay to let them discover that information on their own.

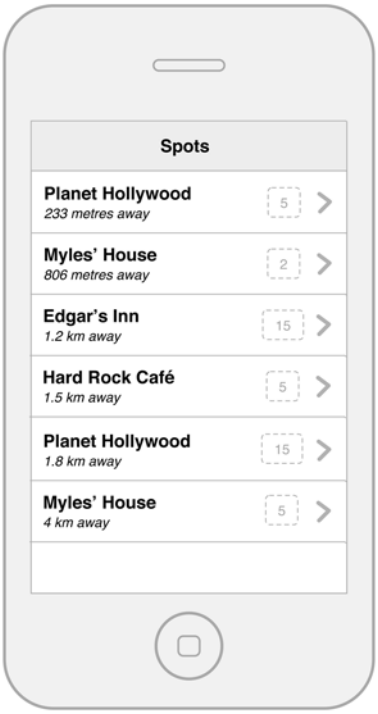

Figure 2.7. A wireframe of StarTrackr's "Spots" listing

Makes sense, right? The only problem is that to make this real, we need to know the location of our users. There are a couple of ways to go about this. The JavaScript Geolocation API exposes the functionality we need to access the device's location sensors, as many smartphones include a GPS. So, we can merely place a button in our interface that starts that process. Unfortunately, the Geolocation API isn't available on all mobile devices, so we'll need a fallback of some sort.

The addition of a fallback text field to our geolocation button has the advantage of giving our users the flexibility to modify their location if they so wish. Why would they want to do that? We'll go into the nitty-gritty of the implementation of the Geolocation API a little later on, but it's worth knowing that it's sometimes a bit like magic. Each device, and indeed each browser, has a different

way of figuring out a location; information can sometimes be unreliable or inaccurate, so allowing a user to refine their position can be a useful feature. Furthermore, if we limited ourselves to using just the Geolocation API, it would mean that users would be limited to only exploring their actual physical location; so if they wanted to explore the hot spots in a different city, they'd have to actually go there!

Figure 2.8 shows the standard approach to asking a user for their location.

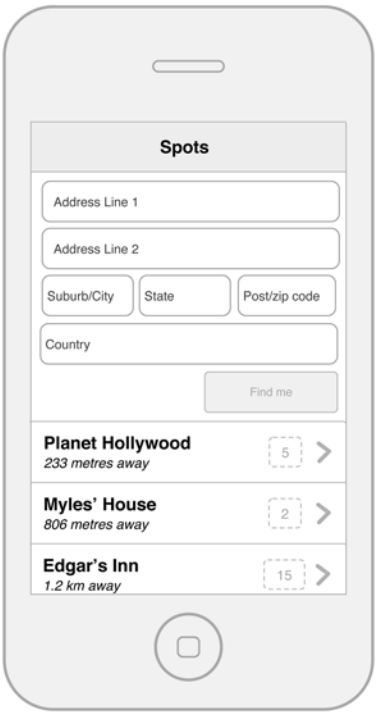

Figure 2.8. An excessive address form

That's not very friendly, is it? Entering data on a mobile device (especially one without a physical keyboard) is much harder than through a keyboard and mouse combination, so if we can reduce the effort our users have to make, we should.

A much better option—even outside of mobile context—would be to let our users enter information in a free text field, and then infer their location using a third-party service or API (we'll look at how to hook this up in Chapter 5). The canonical example of this kind of field is the Google Maps search interface.

Figure 2.9 shows an example of a simpler location entry interface.

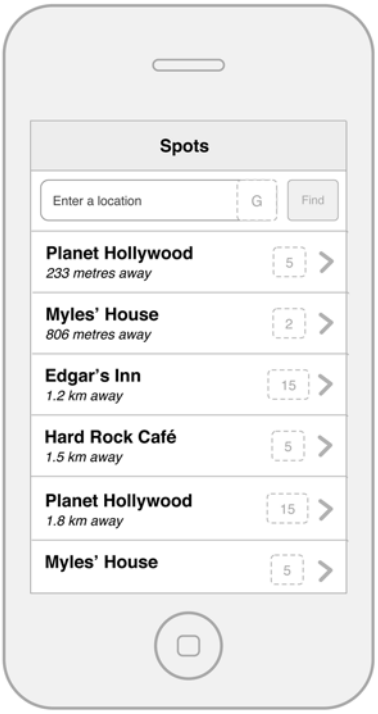

Figure 2.9. A simpler approach

That's much better!

This brings us to an important (and easy to overlook) consideration for touchscreen devices. The available area for our form changes, depending on whether or not the keyboard is displayed. Figure 2.10 shows a minimal form on an iPhone, with the keyboard active.

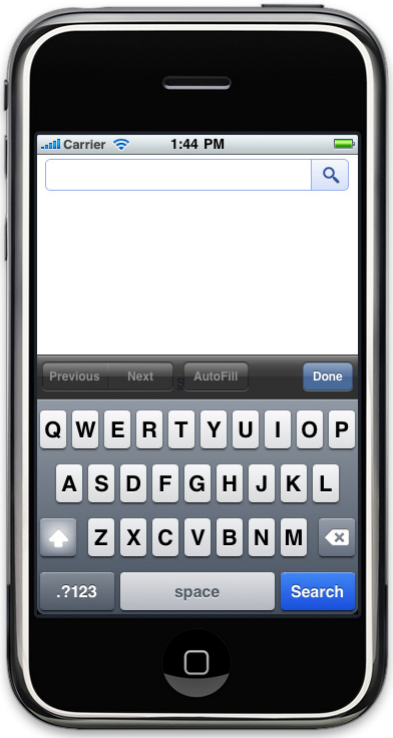

Figure 2.10. The on-screen keyboard can obscure a significant portion of a page

Our once blank canvas is now half-filled with a keyboard. In fact, in portrait mode, the iOS keyboard takes up 260px of the 480px space we have to play with, reducing our usable space to less than half of what we started with. When building forms for mobile devices, always keep this in mind, and make sure you're not hiding essential information or controls behind the keyboard.

Overview and Detail

So, if we're happy with our list structure, we can move on to creating what a detailed view for a given location will look like. This is an example of the master/detail pattern we touched on earlier. From the index of locations, our users have selected an area of interest. Once they click on that location, we can narrow the focus and show them information from that one spot. Let's take a stab at that—Figure 2.11 shows what we've come up with.

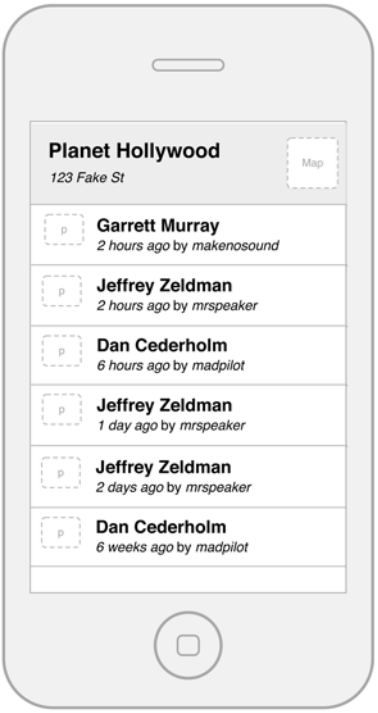

Figure 2.11. A wireframe for the detailed view of a spot

All the vital bits of information about a spot are present: the name, address, and a breakdown of the recent celebrity sightings that have occurred there. If there's a photograph associated with a sighting, we can include it as a thumbnail (and perhaps link to a larger version). There's plenty more we could include here—for example, a breakdown of the most-sighted celebs at this particular spot, or a perhaps a photo gallery of each sighting—but we're trying to keep it simple, right? The set of information we're showing here will let our users know who's been spotted recently—the most important piece of information we can provide them with.

Finding Sightings by Celebrity

We've now worked out the location-centric path through our application, but what if our users are only interested in a particular celebrity? Charlie Sheen showing up at their local bar every evening is irrelevant if they're only interested in what Brad Pitt is up to. Again, a master/detail view is the pattern we are going to use here, as it makes sense in this situation. An index of names ordered alphabetically is a familiar and easy-to-grasp interface for most users (mobile or otherwise). Figure 2.12 shows our wireframe for this screen.

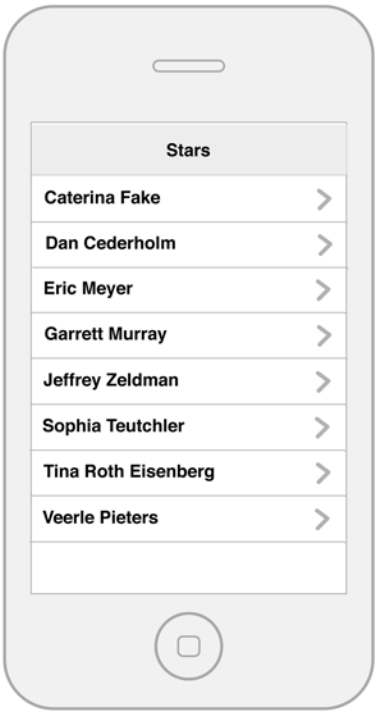

Figure 2.12. Wireframing the Stars listing

There we are: a dead-easy way for our users to find their celebrity of choice. It's worth noting that such a basic interface is potentially problematic; if our app becomes very popular and there are suddenly 1,000 stars in our database, a list like this becomes unwieldy.

It'd be fine if we were only interested in the Brad Pitts of the world, but if our taste changes to, say, Zach Braff, that's a whole lot of scrolling to do. The Android contacts application addresses this issue by including an additional scroll tab, allowing users to jump around the whole list (with lesser accuracy). iOS has an extra alphabetical scroll area that can be used to skip to the start of a single letter grouping. Both those platforms also let you narrow the list by searching within it. For our application, we're going to start simple, and deal with that problem if and when it becomes an issue for our users.

That's our master view. Now on to the detail for a single star, which you can see in Figure 2.13.

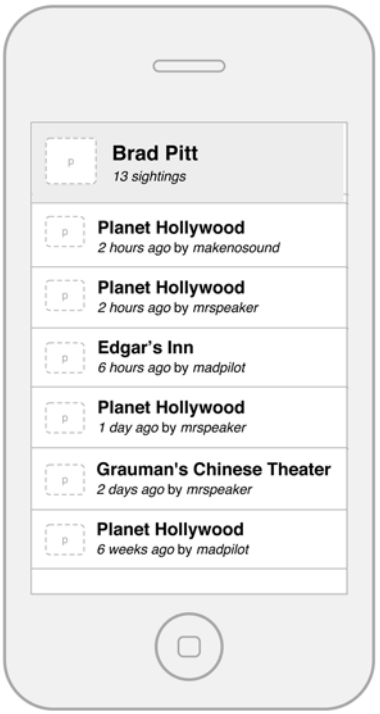

Figure 2.13. Viewing a single Star

The detail view for a star is basically the same as for a location, except the roles are reversed. Again, we have all the bits we need: the name of the star we selected, and a list of recent sightings with their locations. You can think of these pages as an index for sightings filtered by the context through which we arrived: one way is through spots, the other through celebrities.

Adding a Sighting

Now that we've catered for user movement through our application, we need a way for users to contribute to our content. The success of our application is going to depend on the amount of information we can provide people with. Imagine a café review application that has no cafés—what's the point of a celebrity-sighting service if no one can track their sightings?

Encouraging users to fill out forms online is difficult at the best of times, but it's exacerbated on mobile. The longer it takes for users to fill in a form on their mobile device, the higher the possibility that failures or errors will happen—say, losing network connectivity. Designing for efficiency is paramount.

We have to make sure we're including everything necessary to flesh out a sighting. A sighting is the combination of a star at a spot on a given date. Those three facts are all we really need.

Figure 2.14 shows a basic mockup for this form. It strikes a good balance by collecting all the information we need, while still being simple enough to not scare users away from using it.

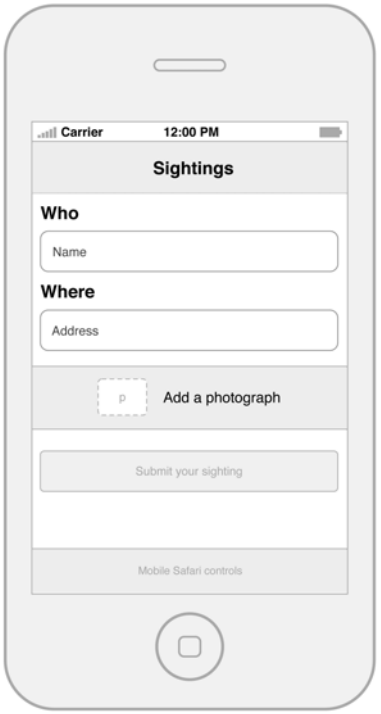

Figure 2.14. Adding a Sighting

Tying It All Together

So, we have all the components of our application in place. The problem is they're separate contexts, so we need a way for our users to flick between them easily. Luckily for us, there's a tried and tested solution out there that we can use—the tab bar. Let's review some of the features of the tab bar:

- establishes the structure of an application
- lets users easily move between an application's sections
- indicates the current state/location of the app

Sounds right up our alley! The flat hierarchy of the tab bar actually suits us here; we have three main sections that are of equal importance.

However, the implementation of a tab bar needs careful consideration, especially when thinking about a mobile web application. In a native app, the tab bar is usually a fixed element at the top or bottom of the screen. Inside the browser, though, this is trickier to accomplish. So what are our

options? Placing the tab bar at the base of the viewport would mean placing it directly above the browser controls.

For a mobile web application, it makes more sense to place the tab bar at the top of the viewport, for the result shown in Figure 2.15.

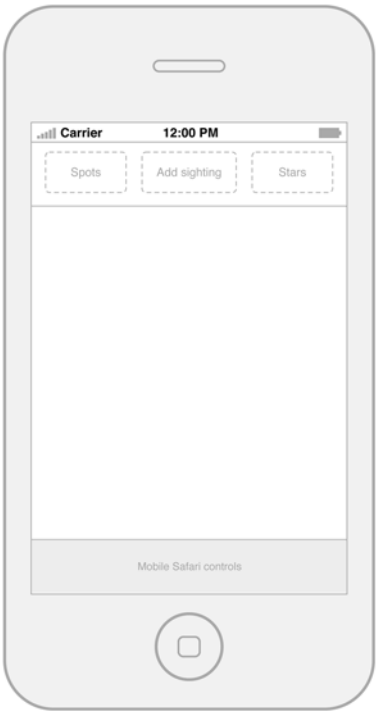

Figure 2.15. Placing the tab bar at the top makes more sense

This improved placement has the added advantage of fitting the expectations of Android users, since tabs on that platform are generally at the top anyway. If we roll the tab bar out to each of the screens we've looked at already, we can see our application is starting to take shape. There's a consistent, flexible structure, and our users can easily identify which context they're in.

The Fix Is Out

Whether it appears at the top or bottom of the screen, the tab bar is often fixed in position, with the main content of the application flowing underneath it. This presents a couple of problems, the most obvious being that none of the current crop of mobile browsers support `position: fixed` in CSS. There are some JavaScript solutions that we'll look at Chapter 6, but they can be complicated (particularly across browsers and platforms), and can also cause performance problems. We need to determine whether the user experience benefits are worth this trade-off in performance. Do our users need access to each section of the app at all times? Will it be confusing if they don't have it?

As a general rule, erring on the side of simplicity is the best way to go. For the in-browser version of our application, we're going to let the tab bar scroll offscreen; users can simply scroll back to the top to change contexts.

Home Screen

On iOS, we have an extra mode that sits somewhere between a web application and a native application. iOS lets us set an option to run Mobile Safari in full-screen mode when our users bookmark our application on their home screen. We'll go into the details of implementing this behavior in Chapter 3, but for now, all we need to know is that full-screen mode removes all the browser chrome. Let's spend a bit of time designing a modified look and feel for this standalone mode. This gives us the opportunity to make our site conform more closely to the iOS user-interface paradigm.

None of the other platforms we're looking at offer this function, so when we're in standalone mode we can conform to the "iOS way" and move the tab bar down to the bottom. It's nice and close to our users' thumbs, as you can see in Figure 2.16.

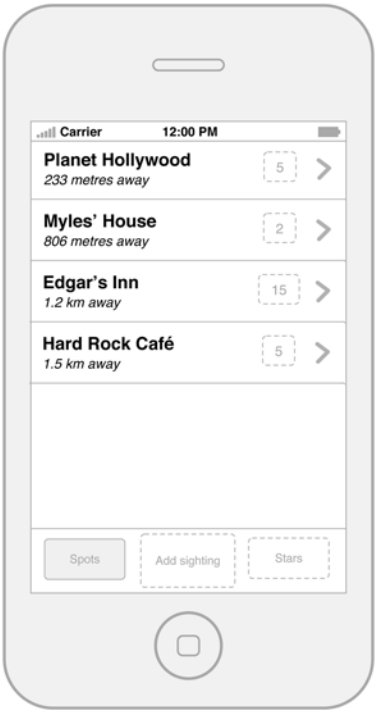

Figure 2.16. Wireframing the "standalone" mode of our app for iOS

That all sounds wonderful, right? And it is, but it's not without problems. The main design gotcha we need to think about at this stage is the lack of the standard browser controls. Our navigational model is drill-down, with the user's options being to move deeper, shift back up a level, or start at the top of the tree again. Without a back button, however, one of those options is impossible; there's

no way for our users to go back, so we'd be forcing them to retrace their steps whenever they wanted to move up a level in our navigation. That means we have to include our own back button somewhere in our standalone mode interface. iOS users expect the back button in the top-left corner, so that's where we'll put it.

Figure 2.17 shows our full-screen wireframe with the addition of an iOS-style navigation bar and back button.

Figure 2.17. Adding a back button completes our standalone mode

Phew! Now that we're happy with the structure and flow of our application, let's attack the fun part: the style and aesthetics of our application.

Establish a Style

Finding a theme to riff on can give us a clarity of vision about the look of our app. In the case of StarTrackr, we can start by thinking about the sort of imagery we associate with celebrity. Here are a few ideas: money, the red carpet, sunglasses, flash photography, film, movies, music, awards shows, the Oscars, stars, gold, diamonds, paparazzi, glamor.

Which items on that list stand out as the strongest images? The colors. The red carpet is synonymous with celebrity, while the color gold brings to mind wealth and glamor. Luckily for us, those colors

should work well together, so we can create a palette with which to build our design. What we've come up with is shown in Figure 2.18.

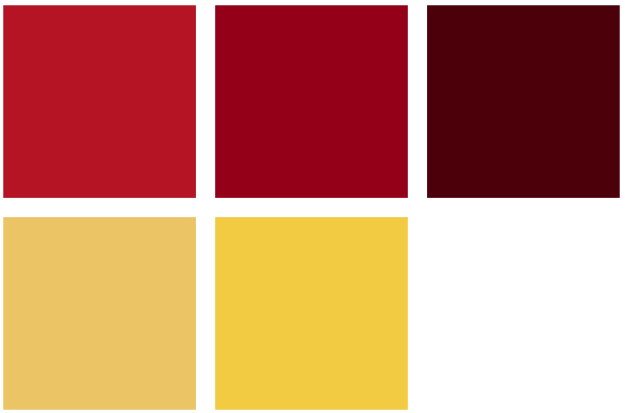

Figure 2.18. A color palette for our app, based on ideas of celebrity glamor

Of those colors the reds are much richer, so they're a more appropriate base for our application. A nice way of testing out multiple color choices is by putting together a basic *in situ* comparison. Let's look at one example, in Figure 2.19.

Figure 2.19. Playing around with colors in the context of our app

It gives us a sense of which colors work best together, and which ones clash. With the colors we've chosen, notice that even though they're completely flat, the darker red gives the impression of sitting *behind* the bright red. We can use this natural depth to our advantage, giving an implied structure to our interface without resorting to over-the-top effects.

Touchable Interfaces

Touchscreen devices let users interact directly and physically with an interface, so we want to build a design that *feels* touchable. One of the secrets to creating beautiful interfaces is to steal some tricks from the real world. Real objects have presence and volume, and their surfaces have variations in light and texture.

They also have a consistent light source. Nothing stands out more in a carefully crafted interface than a shine or shadow that's coming from the wrong direction. Always keep in mind where the light is coming from. We have some leeway—the angle of light doesn't need to be identical for all elements, and in some cases you'll need to change the direction subtly—but in general, we'll try to make sure our light source appears to be coming from above our interface elements. That means shine on the top, shadows on the bottom.

It's useful to imagine the objects in our interface sitting in the same room; that way we can understand how the light might affect them based on where they're located. We're going to follow convention a little here and build up an iOS-like tab bar, so let's have a look at the finished product and the various lighting effects used to build it up in Figure 2.20.

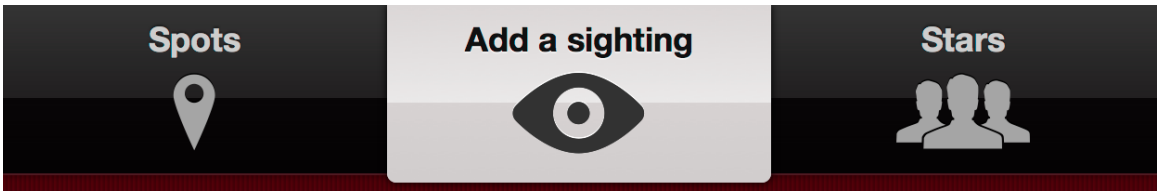

Figure 2.20. The style we'll be aiming for with our tab bar

Here are the key features of our design:

- light source from above
- gradient overlay to give the bar a round, physical presence
- highlight from above
- disappearing into shadow at the bottom
- hard line in the middle indicating the apex of the curve (when compared to the light)
- simple icons

That's our tab bar in its default state—but in fact it'll never look like that in use. Our users will always be somewhere in our application, which means that one of the items will always be selected. How might that look? Figure 2.21 gives an idea.

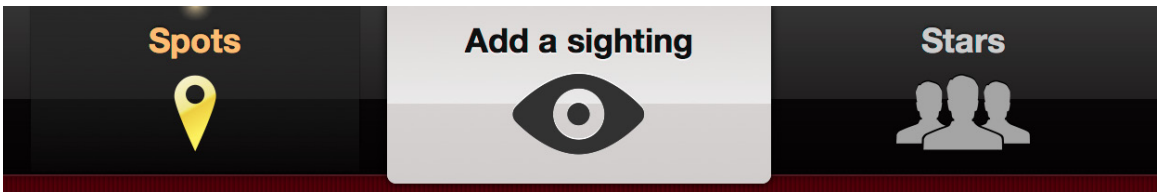

Figure 2.21. Our tab bar again, this time with an item selected

Here's a breakdown of the key features of the selected state:

- dark overlay to increase contrast on the selected button
- use of a contrasting color, in this case the gold color from our palette

- a subtle spot to draw attention
- added depth—highlights and shadow—to our interface icon

The task of styling the tab bar requires a delicate balance. We want to separate it from our content without having it disappear into the background completely. By using muted colours—or rather a lack of color—we're making the distinction between menu and content clear.

We're also doing something a little out of the ordinary with our tab bar. Usually, each item in the bar has the same style and weighting. This assumes that each of those items is conceptually similar, but in our case one of them isn't. "Spots" and "Stars" let our users view content, while the "Add a sighting" item lets them *add* content. We've played on that distinction and given the "Add a sighting" button a contrasting style in an attempt to encourage our users to contribute. There are a number of examples of native applications that use this pattern to draw attention to the function items in their tab bar, as shown in Figure 2.22.

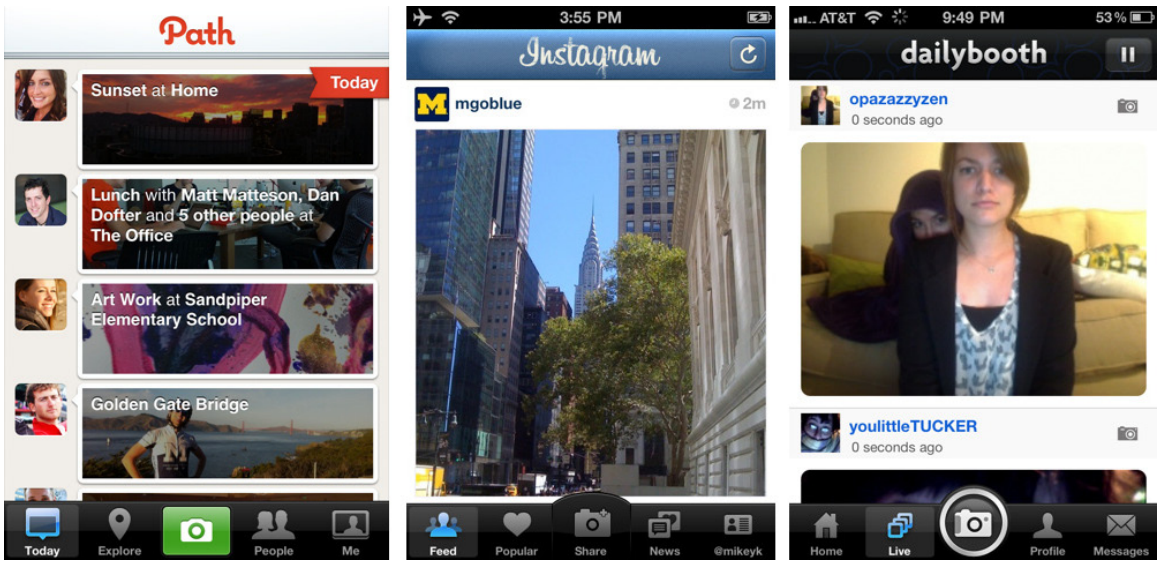

Figure 2.22. Examples of tab bars with differentiated buttons, from Path, Instagram, and DailyBooth

Interface Icons

In our tab bar, we're using icons alongside text to reinforce the function behind each menu item, as well as adding graphical flair to the design. Both Android and iOS have similar design philosophies for their interface icons—namely that they're simple monochrome icons. This makes it easier to create a consistent iconography, and lets us use color in the icons to indicate selected (and other) states for each tab.

It's most important to find icons with the right metaphor to represent your content. As one of the tab bar items in our app is "Stars," using an actual star as that tab's icon would at first seem to make perfect sense. Yet, when you ponder over it more, problems begin to surface. The star has traditionally

had a specific use in many applications: favorites (or "starred" items). If we used a star here, we'd essentially be asking our users to ignore their previous expectations. It makes more sense to use a closer representation of the actual content. In the case of "Stars," that content is a list of celebrities, so using human silhouettes is a better fit.

The other icons for our tab bar are a little easier to decide on. Spots are locations, so some sort of map-related symbol makes sense. We're going to use a Google Maps-style marker, as it's simple and recognizable. For sightings, we're trying to suggest the act of seeing a celebrity. We could use cameras, glasses, binoculars, or autographs—but they're all either hard to make out in the form of a tiny icon, or, as with stars, have associations with other common functions. An eye icon is the simplest and easiest to understand for that function, so that's what we'll use.

"But," you protest, "I'm terrible at drawing—my magnifying glasses always end up looking like spoons! Is there somewhere I can access icons?" Yes, there is! Recently, there's been a proliferation of great royalty-free interface icon sets released, making it incredible easy to find a consistent iconography for an application. Some of the most useful are:

Glyphish, http://glyphish.com/
: A set of 200 or so icons that are available as a free set with attribution requirements, or US$25 for the Pro version.

Helveticons, http://helveticons.ch/
: A lovely set of up to 477 icons based on the letterforms of the typeface Helvetica Bold. Costs are between US$279 and US$439, depending on the set.

Pictos, http://pictos.drewwilson.com/
: Pictos is actually three separate sets consisting of 648 icons! The vector packs for each set are between US$19 and US$29, and are an absolute steal at that price.

A Pixel is no Longer a Pixel

We can no longer assume that we're designing for a single resolution. The iPhone 3GS is 163ppi (pixels per inch), while the iPhone 4 is double that at 326ppi; the iPad is lower at 132ppi, and the Palm Pre sits around the middle at 186ppi. Android's developer guidelines split its devices into three categories: low density, medium density, and high density screens.

What that means for us is that we need to design in high resolution. More and more you'll have to create resolution-independent interfaces, so you can save yourself an enormous amount of time by building designs that way from the start. This doesn't necessarily require you to throw out your favorite bitmap tools and start jumping into a vector world, but it does mean thinking about the end product throughout the design process. For example, instead of creating gradients in Photoshop using, say, bitmaps layered on top of each other, you're better off making those effects using layer styles. That way you can resize those elements without losing definition.

Typography

Typography is central to good web design, yet it's a skill that's often glossed over in favor of shinier features. We're trying to communicate with people, and most of the time that means presenting information or options for them to read and then act upon. Text is user interface. Not only that, but a judicious choice and use of type can give your application meaning.

Unfortunately, the typographic landscape in the mobile space leaves much to be desired. For us web designers, this is nothing new: we're used to finding creative ways to work the limited palette of web-safe fonts. That palette is unfortunately significantly smaller in the world of mobile devices, but it's no excuse to eschew typographic discipline. There are exceptions to this rule, but many mobile devices will have an extremely limited choice of fonts, perhaps only one or two—and those fonts are often not the same from one platform to the next.

Your `@font-face` is Showing

Or rather, it's not. The world of web typography is an exciting place at the moment, mostly due to support for embedding additional fonts via the `@font-face` CSS rule. Whilst we'll avoid going into the specifics of `@font-face` here, we'd be remiss if we failed to mention the support for it in the mobile space. `@font-face` embedding is possible in most WebKit-based mobile browsers, and Opera mobile—though older versions of iOS require SVG fonts in order to work.

The biggest downside to using `@font-face` in a mobile application is performance. Font files are frequently very large; even using a subset of a single weight of a typeface can add an overhead of 100KB of data. For a project that's targeting mobile users, that's often too much to ask. Speed is a feature, so creating a bottleneck in the pursuit of a little extra flair often makes little sense.

Performance Considerations

It's easy to get carried away when designing a mobile app. There are so many examples of stunning interface design on all the native platforms, that it's hard to not want to compete. Alas, we need to be careful about how we use effects in our design. If we create an interface that requires lots of images to implement in HTML and CSS, we may run into performance problems, particularly in scroll-heavy applications. This is one of the unfortunate trade-offs between a native application and a web app. In a native application, we can create beautiful hardware-optimized and accelerated graphical effects using the drawing APIs available in the various native SDKs. The drawing tools available to web apps are a bit more limited. As much as possible, we should be thinking about designing interface elements that we can implement using mostly CSS.

"But we just made a beautiful tab bar! Do we have to throw that all away?" Well, no. Having some heavier graphical effects for the tab bar is no big deal. It will either remain on-screen in a static position all the time, or drop off the page promptly once a user scrolls down. The rows of our main lists, on the other hand, are elements that we should try to keep simple. We can still make them look beautiful, we just have to be a little more inventive. For our list rows, we're going to start with

a solid color for each row, but using a slightly different shade for alternating odd and even rows, as Figure 2.23 shows.

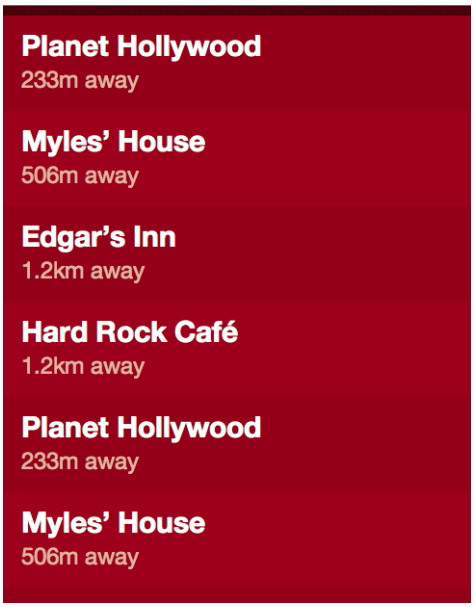

Figure 2.23. A basic list made up of solid alternating colors

Looks a bit flat, right? Let's give those rows a little "pop" with some lighting effects. The easiest way with an interface element like this—a multi-item list with content that could potentially vary in size—is to add a highlight to the top edge and an inset shadow to the bottom edge, as in Figure 2.24.

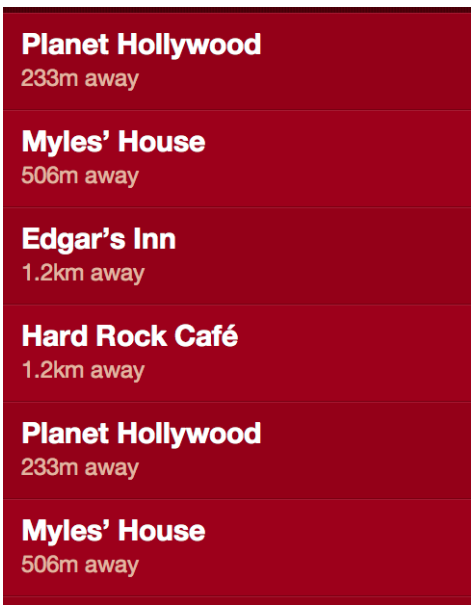

Figure 2.24. Adding a bit of depth to our list with highlights and shadows

That's much better! It's a subtle effect, but it really helps lift the interface and lets our users focus on the content. Plus, it can be implemented using only CSS border properties, so it passes our performance requirements. The problem now is that our text feels a little flat—because it appears to be unaffected by the same light that's adding the highlights and shadows to the rows of our list. Thankfully, we can use the same visual trick to give the text a little more physicality, as Figure 2.25 demonstrates.

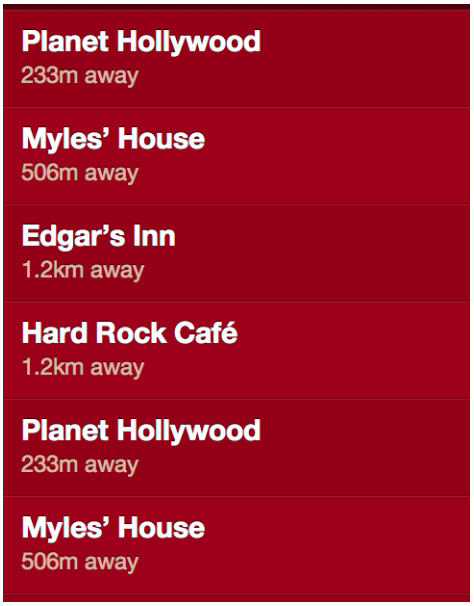

Figure 2.25. Adding shadows to our text gives it a stronger presence

Here we're just adding a one-pixel black shadow to the top edge of our letterforms. This gives the impression that the type has been physically stamped into the interface, the way a letterpress imprints onto paper. It's a simple effect, but it needs to be used carefully: if the text to which we're adding a shadow is too small, it can lose readability.

Remember, subtlety is one of the pivotal points to keep in mind for interface design, as Cameron Adams once said:[6] "Your maturity as a designer can be measured by the subtlety of your drop shadows."

Truer words were never spoken. A subtle interface lets the content of our application speak for itself. Using cheap and over-the-top effects will only distract our users from what we're trying to tell them. Our interface should define a structure for our content and help our users understand the function of our application. Great design is functional, not flashy.

[6] http://www.themaninblue.com/

Testing Design

It's impossible to design a great mobile application without seeing and "feeling" your interface on an actual device. There are some great tools for previewing your artwork on a mobile device. One of the most useful for the iPhone is LiveView (not least because it's free).[7]

LiveView is a remote screen-viewing application intended to help designers create graphics for mobile applications. The app is split into two parts: a screencaster application that you can run on your desktop machine, and an iPhone/iPad application that you can run on your phone. The screencaster lets you position a "window" over your desktop interface, and then broadcasts whatever graphics are under that window to your phone via Wi-Fi. It will even pass clicks from the device back to the screencasting computer. Unfortunately, LiveView is an OS X-only application, but there are similar remote desktop applications for Android and other platforms as well, which should be easy enough to find with a search through your marketplace of choice.

However you decide to do it—saving images to your device, or perhaps creating some image-only web pages—testing your designs and prototypes on your target device is an absolute must.

Reviewing Our Design

Alrighty, let's take a moment and assess where we're up to. We have a structure for our pages and a flow for our application; we have a theme we want to go with, and we've looked at some techniques we can use to give our design some flair. Let's roll out the design to the screens we've created.

Most of the pages use the techniques and structures we've outlined already, but lets break down a couple of these designs and look at them in further detail. First up is the Spots index, shown in Figure 2.26.

[7] http://www.zambetti.com/projects/liveview/

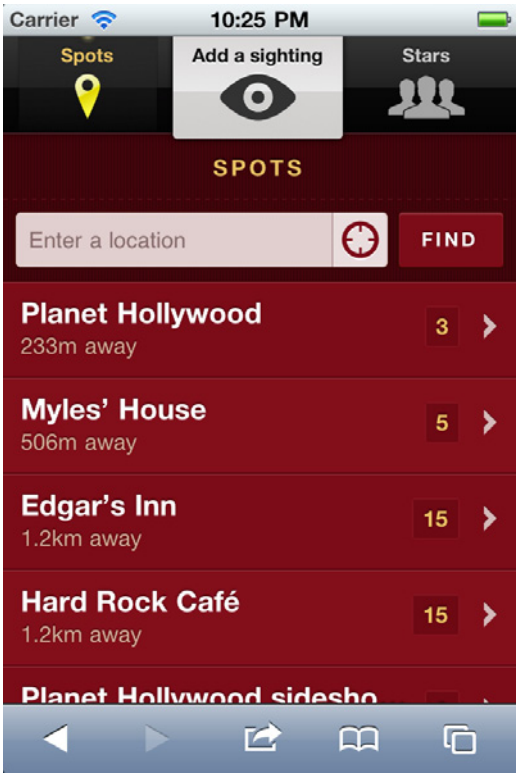

Figure 2.26. The complete Spots index page

Here's a quick review of the elements contained in this page:

- actionable list items have an arrow that suggests more content
- actionable list items have neutral text color, with the yellow reserved for hyperlinks
- a dark background with a lighter foreground
- a "find" button with the same highlights we're using for list rows
- a button for geolocation tied into the text field (this suggests those functions are linked—which they are—and reduces the number of separate elements in the form)

Figure 2.27 shows what the Spots listing will look like when scrolled down to the bottom.

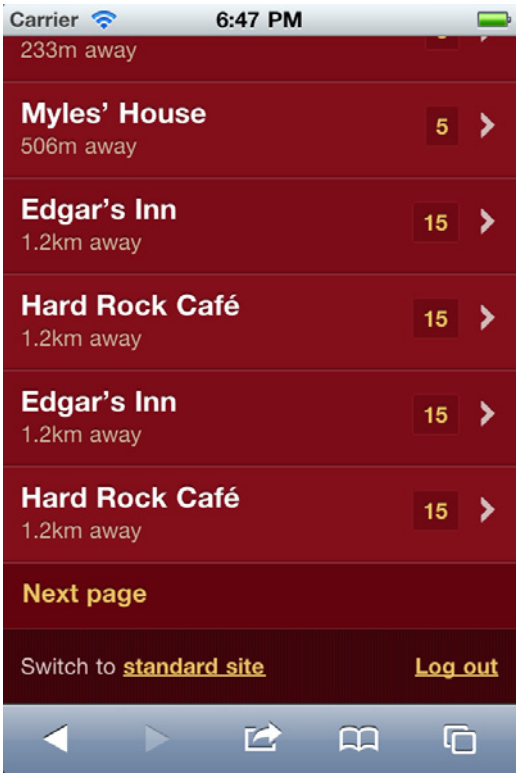

Figure 2.27. Spots index page, scrolled to the bottom

Some points to note here are:

- The footer at the bottom includes links to the full (non-mobile) site, and the ability to log out.
- The "Next page" link (because we don't want to load endless amounts of data over the mobile network, especially when those results might be irrelevant for our users, as the information is potentially old and out of date).

Next up is the detail view of a Spot, shown in Figure 2.28.

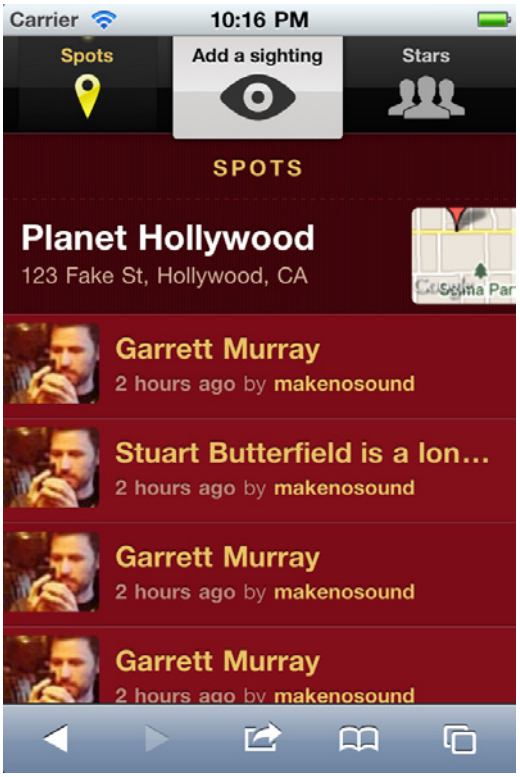

Figure 2.28. Viewing a Spot

In this design, notice:

- the main details are over the background, enabling the main content to sit in the foreground area
- that rows don't have to be limited to a single action (we're using a contrasting yellow for standard hyperlinks)
- a small thumbnail of a map (while practically useless at this size, a map is still instantly recognizable, so we can use it as an icon linking to a real map)

The final design to review is the full-screen or standalone mode. Let's look at our Spots index in standalone, as shown in Figure 2.29.

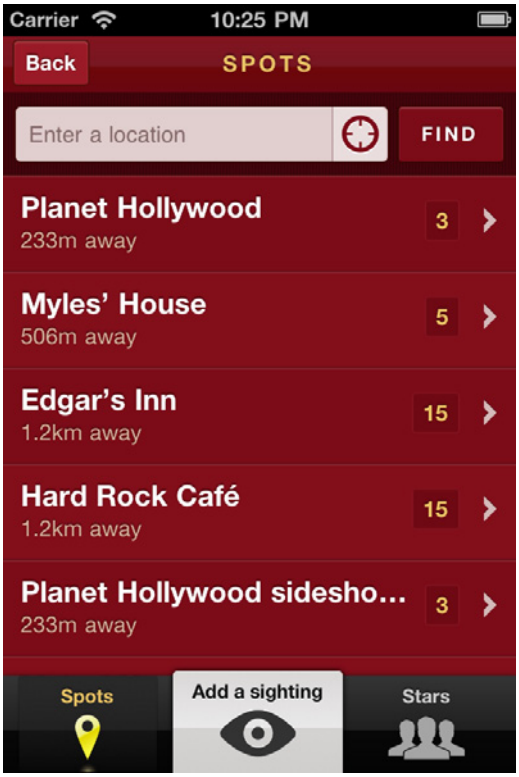

Figure 2.29. The full-screen mode of our Spots listing

All the changes we made in the wireframing stage are reflected here, notably that:

- our navigation is at the bottom
- we're bringing in a header element, which establishes the context and gives us room for more controls (such as our back button)

Application Icons

Now that we've built up our lovely design for StarTrackr, we'll need an icon to match. Our icon will eventually be used in a number of places, but primarily it'll end up on our users' home screens. In all likelihood, it'll be the part of our application that people will see most often, so we need to create a memorable icon that reinforces the function we're offering. In general, good application icons do one or more of the following:

- build from an existing brand iconography
- encapsulate the functionality of the application in a single image
- play off the name to reinforce the identity

Good examples of the first approach can be seen in the Facebook iPhone application, as well as the various mobile applications offered up by 37signals—seen in Figure 2.30. Facebook uses a white "f" on a blue background; it's incredibly simple, but the association with their standard logotype and the color scheme of their website is immediate and strong.

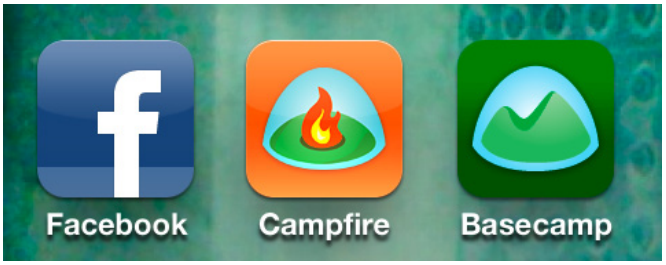

Figure 2.30. The application icons for Facebook, Campfire, and Basecamp

37signals have done a great job creating a set of consistent yet unique icons to match their suite of applications, and this applies to the icons for the two mobile applications they currently offer: Campfire and Basecamp. Campfire is a native application, and Basecamp a web app, but they share a consistency of shape (and thus brand) while staying different enough to be recognizable. The strong base color helps differentiate each icon from the other.

There are some excellent examples of this approach all over the place, particularly in some of the default applications on iOS, as you can see in Figure 2.31.

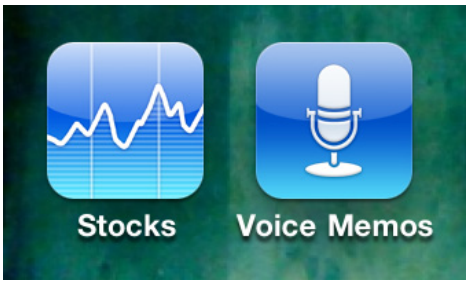

Figure 2.31. The icons for the Voice Memos and Stocks apps on iOS

Voice Memos, for example, uses an old-school microphone to make the connection between the application and its purposes. The iPhone Stocks app uses a graph to indicate the function—keeping track of ever-changing stock prices—and hint at the application's interface.

Unfortunately for us, StarTrackr is yet to have the same brand recognition as, say, Facebook, and the functionality of our app—finding and adding celebrity sightings—is difficult to distill in a single, comprehensible image. Instead, we're going to keep it uncomplicated and just use a star as the main graphic element for our icon. This gives us a strong image that helps reinforce the name of our application, and it has a nice association with the notion of celebrity. This concept also lets us build on the visual style that we've established with the application interface.

In our design to this point, we've been looking at the idea of the red carpet and trophy gold, awards, and glamor we associate with celebrity, and we can keep that flowing through the application icon. Figure 2.32 shows what we've come up with.

Figure 2.32. The application icon for the StarTrackr app

Icons should be, well, *iconic*, and our star fits the bill perfectly. Sure it's simple, but that's exactly what we're after. Some general rules when designing applications icons are:

- They should be forward-facing. We want volume and presence, so some perspective is nice, but don't go overboard.
- The light source should be top-down.
- Fill the entire background. While some platforms support transparent images, your app will look out of place on iOS without a complete background color.
- Consider the icon's appearance at small sizes, and create one that is recognizable at every resolution.
- Create artwork in a vector format. We'll see later on that there are a myriad of sizes for our icons.
- Subtle texture will make surfaces feel more real and less clinical.
- Above all, keep it simple.

Ready to Shine

That's it for the design of our application. We have a strong theme with a consistent feel, and a layout that's familiar to our users that establishes a navigational structure for the rest of the experience. Now let's learn how to make it real.

Chapter 4

Mobile Web Apps

Now that we have a beautiful mobile website up and running, it's time to give it that extra touch of interactivity. Our client wants more than a simple mobile version of their website; they want an application on par with anything the app marketplaces have to offer.

It's a tall order, but device providers have given us a tantalizing set of capabilities that not only allow us, but encourage us to reproduce native behaviors in a web setting. If we play carefully with this double-edged sword—that is, knowing our limitations and exploiting our strengths—we can transform our websites into full-fledged apps that are a joy to use.

Setting up Shop

As a first pass for *app-ifying* our mobile website, we'll set up our environment, hook up a couple of common DOM events, and use the native-like features gifted to us for some gratifying "quick wins." These should help us get well on the way to creating a usable app that we can build on with more advanced features.

Frameworks and Libraries

First of all, we need to make some tough decisions. The mobile web app development landscape has no yellow brick road; it's more like strolling through a dense, twisted forest of super-cool devices and browsers—all with wildly varying capabilities. To help us out, new frameworks and libraries for mobile app development are sprouting up like mushrooms.

A framework can greatly simplify our task by taking care of cross-device and cross-browser inconsistencies, and offering prebuilt UI widgets and designs that we can patch together as an application. Sencha Touch[1] and jQuery Mobile[2] are two big players, but there are scores of others, and there is certainly no "winning" framework at the moment. All the big frameworks have benefits and drawbacks, so you'll need to carefully test and evaluate the options to see if they meet your needs for performance, compatibility, and customization. In the interests of teaching you the underlying concepts of developing for the mobile web, we'll be writing our own UI code in this book.

Okay, but how about a DOM library? Surely there's no need for us to write our own DOM manipulation code, right? Over the last few years, JavaScript libraries have changed the way we work with the DOM. We can modify and animate our documents with some terse, elegant APIs, and maintain some level of confidence that we won't be spending the majority of our workdays troubleshooting inconsistencies across desktop browsers.

So, does it make sense to bring this convenience to mobile? The answer is, of course, "it depends." Most of the major DOM libraries take great care to ensure they function across all the desktop browsers, including IE6. But when working on a mobile web app, IE6 is a non-issue. Again—you need to evaluate your options. For the remainder of the book, we'll stick with jQuery as it's very well known, but all of the concepts (and most of the code) convert easily between libraries—or to plain JavaScript.

A Slimmer jQuery

If you like jQuery, but are hesitant about its file size, you might like to check out the interesting Zepto project.[3] It has a jQuery-compatible (though not full-featured) API, but is only about 4k when minimized. This is possible because it only targets mobile WebKit browsers, rather than wasting hundreds of lines of code getting IE6 on board.

Debugging Mobile JavaScript

We've settled on jQuery, so let's add our library to the bottom of the HTML document, as low as you can before the final `</body>` tag. We'll also throw in a quick alert to verify that jQuery is up and running:

ch4/01-jquery.html *(excerpt)*

```
<script src="javascripts/vendor/jquery-1.6.1.min.js"></script>
<script type="text/javascript">
  $(document).ready(function(){
```

[1] http://www.sencha.com/products/touch/
[2] http://jquerymobile.com/
[3] http://www.zeptojs.com/

```
    alert("StarTrackr loaded!");
  };
</script>
```

What if you didn't receive an alert? What went wrong? Well, there are a few ways to troubleshoot and debug. The first is to test it on your desktop browser. Safari or Chrome (or another WebKit-based browser) will do a good job of emulating an iPhone or Android phone, and their built-in tools for debugging will save you a lot of headaches.

A rudimentary console is also available on the iPhone, though it's disabled by default. Switch it on in the Safari preference screen via the general phone preferences. Not only will the console show you any errors that occur on the page, you can also throw your own messages in there—which is far more convenient than spawning alert boxes throughout your code. There are a few different "types" of logs available to us:

```
console.log("General log item");
console.info("Just some information.");
console.warn("Oooh, be careful here…");
console.error("We've got a problem!");
```

Figure 4.1 shows what the output of these logs looks like on an iPhone.

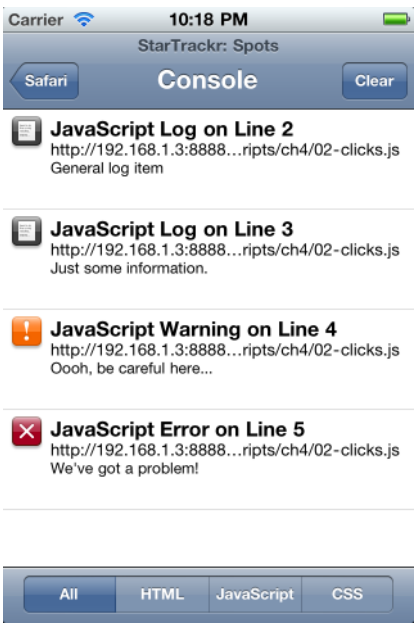

Figure 4.1. Checking out the console on an iPhone

We've just output strings here, but you can also print the value of variables. On the iPhone, you'll only get the value of simple types (strings, numbers, and so on)—not objects. It's still very useful,

though—just remember to remove these messages before you go live. If a user's device lacks a `console` object, it will throw an error and stop your scripts.

If you're testing on an Android device, you can enable USB debugging on your phone, and then use the Android Debug Bridge that comes with the Android SDK to log messages from your device to your computer's terminal. (We'll cover installing the SDK in Chapter 7.) To view these logs, run `adb logcat` from the **platform-tools** directory of your SDK.

Events

For many users, the most noticeable difference between a website and a native application is the way each one transitions between screens or pages. Native applications respond instantly, sometimes sliding or fading content on and off the screen, whereas websites show a few seconds of white screen before the next page's content starts loading in.

There are a number of ways we can make our web app feel more responsive. As you've probably guessed, using Ajax to pull in page components to avoid a refresh is an important component of our strategy. This approach will pay dividends in Chapter 7 when we use PhoneGap to bring our app into the native world.

Before we load in content with Ajax, and use transitions and animations to swoosh things around, we'll need to capture some events. For the most part, handling events on mobile is no different from the desktop, but there are a few key points worth learning about, so we'll start there.

Let's begin by capturing click events on the links in our tab bar. As a quick refresher, here's the markup for the navigation:

ch4/02-clicks.html *(excerpt)*

```
<ul id="tab-bar">
  <li id="tab-spots">
    <a href="#page-spots">
      Spots
    </a>
  </li>
  <li id="tab-sighting">
    <a href="#page2">
      Add a sighting
    </a>
  </li>
  <li id="tab-stars">
    <a href="#page3">
      Stars
    </a>
  </li>
</ul><!-- #tab-bar -->
```

To verify that we can capture click events on each list item, we'll fire off another alert message:

javascripts/ch4/02-clicks.html *(excerpt)*

```
$("#tab-bar li").click(function(e){
  e.preventDefault();
  alert("Coming soon!");
});
```

Running Scripts

All the code snippets that follow will assume the code is either inside, or is called from inside the `ready` function. That way, we're sure that our document is loaded before we run any code that relies on its presence.

We've attached our function to the click event on all the list items, and we've prevented the event's default behavior from occurring, so the browser won't navigate away to the link address. Fire that up on your mobile phone to make sure it's working. It might be a boring alert message, but seeing your code running in the palm of your hand is fairly cool. Figure 4.2 shows the output on the Android browser.

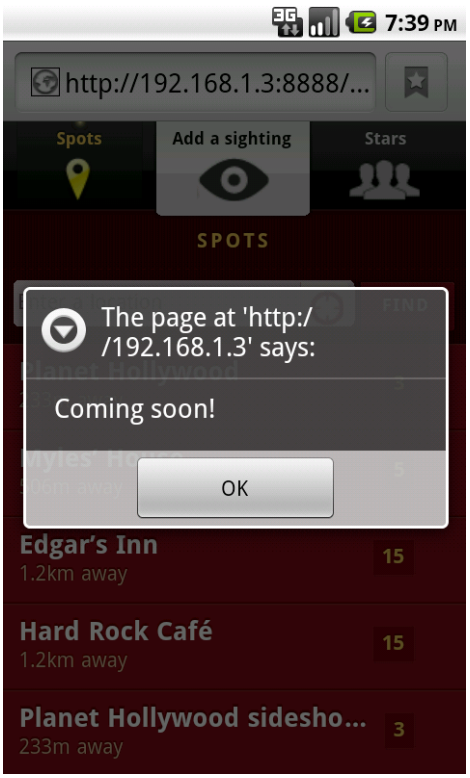

Figure 4.2. We've captured the click event and used it to display a message

The DOM `click` event comprises two separate steps: a `mousedown`, followed by a `mouseup`. Evidently, it's an event designed for a mouse, where clicking takes just a fraction of a second, and is done after the user has moved the mouse into the correct position. This interaction doesn't always replicate so well on the touch screen, and the click can end up feeling a bit sluggish on some devices.

For a slight improvement, only target the `mouseup` event; however, the `mouseup` event is independent of the `click` event, so we still need to stop the click from following the link to another page. Once we've curtailed the hyperlink, we can run our code in the `mouseup` event handler, which we attach using jQuery's `bind` method:

javascripts/ch4/03-mouseup.js *(excerpt)*

```
// Stop clicks from following links
$("#tab-bar li a").click(function(e){
  e.preventDefault();
});

// Do our magic on mouse-up!
$("#tab-bar li").bind("mouseup", function(){
  alert("Coming soon!");
});
```

Simple Touch Events

Although mobile devices can pretend that your finger is a mouse pointer, and fire regular click events, they also generate *touch* events, which are more accurate and useful. However, these come loaded with so many cross-browser booby traps, that you'll start wishing you could go back to developing for IE6!

Fortunately for us, detecting a touch is simple enough:

javascripts/ch4/04-touchend.js *(excerpt)*

```
$("#tab-bar li").bind("touchend", function(e){
  alert("Coming soon!");
});
```

Most recent mobile touch devices will fire the `touchend` event. This fires when the user lifts their finger from the screen after touching it. There are also a few other events that map quite nicely to the mouse equivalents, as Table 4.1 summarizes.

Table 4.1. Touch events and their corresponding mouse events

Touch event	Mouse event
`touchstart`	`mousedown`
`touchmove`	`mousemove`
`touchend`	`mouseup`
	`mouseover`

Notice that there's no `mouseover` event on a mobile device—after all, it's impossible to know where you're hovering your finger! It might be a small detail, but it has some interesting implications for user interaction design, as we discussed briefly in Chapter 2. No hover states or tooltips means the user can no longer move their mouse around the screen looking for help, so you have to make your interfaces nice and self-explanatory.

So far, we've wound up a subset of mouse events—nothing too fancy. In the next chapter, we'll explore some advanced touch features of mobile devices like swiping and gesturing. In the meantime, let's continue our investigation of simple touch events.

Clicking with Feature Detection

Touch events can be more responsive and accurate, but they're not supported on all devices. We may be focusing our attention on the latest generation of super-duper phones, but it's always a good idea to provide support for as many browsers as possible. In addition, relying solely on touch events makes it impossible to test your app on a desktop browser. It's great fun to see your hard work displayed on a phone, but not so fun having to switch between desktop and mobile phone copious times as you're testing features.

You can (and should) use the various device emulators that will interpret your mouse clicks as touch events, but if you're used to desktop web development, being able to test your apps in your desktop browser is very convenient.

To make our code run on both new and old mobile browsers—and desktop browsers too—we'll implement a piece of **feature detection**. Feature detection involves probing the user's browser to see which cool features we can use, as well as (we hope) providing adequate fallbacks for features that are missing.

We can do this by assigning a few variables:

javascripts/ch4/05-touchdetect.js *(excerpt)*

```
var hasTouch = "ontouchend" in document,
    touchEndEvent = "touchend";
```

```
// Default to mouse up, if there's no touching
if (!hasTouch) {
  touchEndEvent = "mouseup";
}
```

If a browser supports touch events, the window's `document` object will have the `ontouchend` event in it. We create a variable called `touchEndEvent`, which initially contains the string `"touchend"`; we replace this with `"mouseup"` if our touch detection turns up nothing.

Now that we have a variable containing the event we want to target, we can bind the variable instead of a static string, and the correct event will be handled based on the device's capabilities:

javascripts/ch4/05-touchdetect.js *(excerpt)*

```
$("#tab-bar li").bind(touchEndEvent, function(){
  alert("Coming soon!");
});
```

Ternary Operator

There's a shortcut in JavaScript (and in many other programming languages) that allows you to set a variable conditionally as we did above. It's called the **ternary operator**. It has the syntax `a ? b : c`, which translates to: if `a` is true, return `b`; otherwise, return `c`.

So we could have written the `touchend` assignment above as:

```
var touchEndEvent = "ontouchend" in document ? "touchend" : "mouseup";
```

This reads as "if `document` contains the property `ontouchend`, return `"touchend"`; otherwise, return `"mouseup"`." The result is then assigned to the `touchEndEvent` variable. Ternary operators are a terse (some would argue "cryptic") way to do a conditional assignment, and, as with the traditional programmer battle of tabs versus spaces, you either love the ternary operator or hate it!

Without trying to scare you, there is still more to say on the topic of clicking. Even the simple click has a raft of new problems when moving from the mouse to the finger, one being the implications of the double-click. Your mobile device needs to wait a relatively long time to see if your click is going to become a double-click, and that delay manifests itself as a small but noticeable lag. In Chapter 6, we'll have a look at removing this, with the "quick click." For the rest of this chapter, though, we'll err on the side of simplicity and rely on the good old `click` event.

Quick Wins

As we move though the book, we're going to meet some relatively complex code for recreating native effects and behaviors. Thanks to some (fairly) standard hooks and APIs, however, there are

a few tricks we can employ to add a bit of pizzazz to our apps without having to do much work at all.

Nifty Links

For security reasons, mobile web applications are sandboxed away from many built-in features of the mobile device; for example, unlike native apps, they're not able to retrieve a list of a user's contacts, or take a photo with the device's camera (yet). However, they do have the ability to open a few core applications and fill them with data just by using some carefully constructed hyperlinks.

Email

The simplest of these is the well-known `mailto:` URI scheme. On the desktop, these will launch your default mail application—and a smartphone does the same:

ch4/06-links-forms.html *(excerpt)*

```
<a href="mailto:feedback@startrackr.com?subject=Complaint">
  Send complaint
</a>
```

This will bring up the device's email app, with the subject line filled out with the variable we passed in, as Figure 4.3 shows.

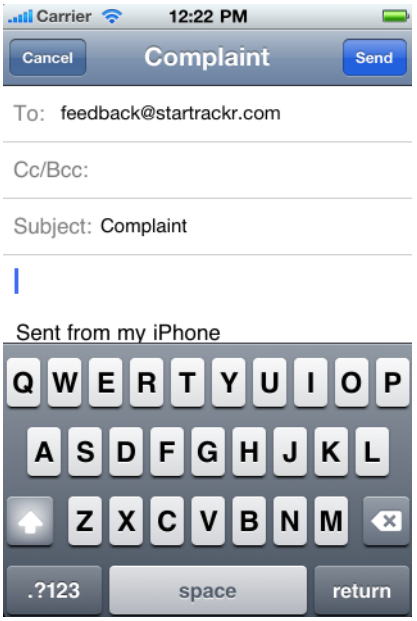

Figure 4.3. `mailto:` links will bring up the phone's email app

Phone Numbers

We can also assist in dialing a phone number using the `tel:` URI scheme. This will bring up (but not dial, of course) a phone number:

ch4/06-links-forms.html *(excerpt)*

```
<a href="tel:1-408-555-5555">
  Call in a sighting!
</a>
```

In fact, for the iPhone, there's no need to even wrap the number in a hyperlink. Mobile Safari includes a feature that automatically detects phone numbers in a page and turns them into links for you. "Excellent," you may think—until the first time it tries to turn your product IDs into "dialable" phone numbers! Similarly, there's a feature that turns address-like text into map links. But like any automagic feature, it's not always what you want. If that's the case, you can include `<meta>` tags in the head of your page to disable these features:

```
<meta name="format-detection" content="telephone=no"/>
<meta name="format-detection" content="address=no"/>
```

Again supported by the iPhone but not Android is the `sms:` URI scheme, which also takes a phone number, but opens up the text message application. At this point you might be worried about the limited support—fortunately, it's not a big problem if a device fails to recognize a URI scheme, as nothing will break; the link will simply do nothing.

Maps

Turning now to maps, the situation is a little less ideal. If you want to open a map and zoom to a given location, there's no widely implemented, standards-defined way to do it. A common method is to simply construct a URL that points to http://maps.google.com/ with a properly formatted latitude and longitude. Both iOS and Android will open this using the built-in mapping application rather than following the link in the browser:

ch4/06-links-forms.html *(excerpt)*

```
<a href="http://maps.google.com.au/maps?q=sitepoint">Visit us!</a>
```

A more standards-friendly version is the `geo:` URI, which accepts a variety of values that will be interpreted as map data by the device's mapping application. You can pass a latitude and longitude:

ch4/06-links-forms.html *(excerpt)*

```
<a href="geo:-33.87034,151.2037">Visit us!</a>
```

Or a street address or business name and location:

```
<a href="geo:0,0?q=123+Fake+St">Visit me!</a>
```

This is certainly nifty, but it's currently only supported on Android.

Form Field Attributes

With our links all polished up, let's turn to forms. HTML5 drags the basic form into the future with a quiver of shiny new input types and form attributes, which are well-supported on the current crop of mobile devices.

The HTML5 `placeholder` attribute of an input field will populate the field with a user prompt, which disappears when the user focuses on it. This is commonly used to avoid the need for a field label, or to offer additional help text to the user:

ch4/06-links-forms.html *(excerpt)*

```
<fieldset>
  <label for="name">
    <span>Who</span>
    <input type="text" name="name" placeholder="Star's name"/>
  </label>
  <label for="tags">
    <span>Tags</span>
    <input type="text" name="tags" placeholder="Tag your sighting"/>
  </label>
</fieldset>
```

The iPhone's keyboard tries to help out users by capitalizing the first letter in a form field. Most of the time, this is what you want—but not always; for example, in the tags field in our sample form. The iPhone will also attempt to correct words it fails to recognize, which can become a problem for our celebrity name field. These features can be disabled via the `autocorrect` and `autocapitalize` attributes:

ch4/06-links-forms.html *(excerpt)*

```
<fieldset>
  <label for="name">
    <span>Star</span>
    <input type="text" autocorrect="off" placeholder="Star's name"/>
  </label>
  <label>
    <span>Tags</span>
    <input type="text" autocapitalize="off" placeholder="Tag your sighting"/>
  </label>
</fieldset>
```

Note that these attributes are nonstandard, in that they're not in the HTML specification—at least for now.

Turn off the Automagic

If the majority of your form's fields require these attributes, you can also add them to the `<form>` tag itself, to apply them by default to all fields in that form. You can then override this setting on any given field as required.

Another HTML5 feature that's useful for mobile sites is the addition of a number of new input types. Beyond the traditional `type="text"`, HTML5 provides email, number, url, date, and even color inputs. These will all display as simple text fields on most browsers, but the iPhone cleverly provides appropriate keyboards for the data in question—for example, including shortcut keys for @ and . (period) when you use `type="email"`. For `type="number"`, it will provide a number pad instead of a traditional keyboard, as shown in Figure 4.4. The BlackBerry browser even provides date and color pickers for `date` and `color` input types.

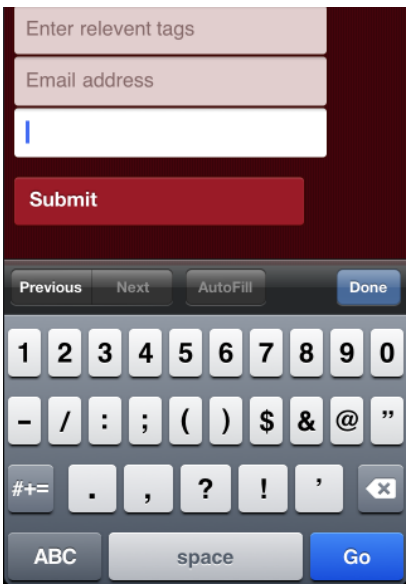

Figure 4.4. A numeric keypad appears when focusing a number input on the iPhone

Here's an example of these input types in action:

ch4/06-links-forms.html *(excerpt)*

```
<label>
  <span>Tags</span>
  <input type="text" autocapitalize="off" placeholder="Relevant tags">
</label>
<label>
```

```
  <span>Number of celebs</span>
  <input type="number" placeholder="Number of celebs">
</label>
<label>
  <span>Tags</span>
  <input type="email" placeholder="Your email address">
</label>
```

Support for all these features is inconsistent, so you'll need to test your target devices. The good news is that even when support is lacking, the app won't appear broken. Unsupported input types will simply behave as regular text fields, and users will be none the wiser.

Because the iPhone was the first to market, many of the iOS proprietary tricks have wide support on other devices; yet, not everyone wants to implement a competitor's features! Thankfully, support for the HTML5 standard is growing, and that should trickle down to all the big mobile players soon.

Loading Pages

Now that we've learned the basics of handling touch screens, and picked up a couple of quick wins by optimizing our links and forms, it's time to roll up our sleeves and get to work on the biggest component of just about any mobile web app. Unless your application is very basic, chances are you're going to need more than a single page, and therefore, you need to think about how to switch between pages. As we discussed earlier, our client is unlikely to be impressed with the web-like page reloads that currently exist in our app, so we need a way to hide the request/response cycle from the user. There are three main ways we can do this:

1. putting everything on one page, and then hiding and displaying sections as required
2. loading in new pages via Ajax
3. including only the complete skeleton of the app up front, and then bringing in data as required

The approach you take will depend heavily on the application. We'll start by looking at the first (and simplest) approach, and load all our content up front. This will let us look at how we can handle transitions between the various states. Of course, for a real app that depended on the user's location to fetch up-to-date information, you'd want to opt for one of the other two methods; we'll look at them at the end of this chapter.

Swapping Pages

If all our content is loaded in a single HTML page, a "page" from the point of view of our application is no longer a full HTML document; it's merely a DOM node that we're using as a container. We need to choose a suitable container and an appropriate way to group our pages, so that our scripts can manipulate them consistently.

We'll start by creating a container `div` (called `pages`), which contains a number of child `div` elements that are the actual pages. There can only be one page visible at a time, so we'll give that element a `class` of `current`. This `class` will be passed to whichever page is the active one:

```
<div id="pages">
  <div id="page-spots" class="current">
    <!-- Spots Index -->
  </div>
  <div id="page-spot">
    <!-- Spot Detail -->
  </div>
  <div id="page-sightings">
    <!-- Add Sighting Form -->
  </div>
  <div id="page-stars">
    <!-- Stars Index -->
  </div>
  <div id="page-star">
    <!-- Star Detail -->
  </div>
</div>
```

This list of pages will sit below the tab bar—so no need to change the markup of our navigation. We have, however, hooked up the links to point to the various sections by way of their `id` attributes; this will let us use a sneaky trick to show pages in the next step:

```
<ul id="tab-bar">
  <li>
    <a href="#spots">Spots</a>
  </li>
  <li>
    <a href="#sightings">Add a sighting</a>
  </li>
  <li>
    <a href="#stars">Stars</a>
  </li>
</ul>
```

After this, we need a couple of styles for hiding and showing pages. In our markup, every page is a first-level child of the main `#pages` container, so we can rely on that fact and use a child selector (>). First, we'll hide all the pages; then we'll unhide the page that has the `current` `class`:

stylesheets/transitions.css *(excerpt)*

```
#pages > div {
  display: none;
}
```

```
#pages > div.current {
  display: block;
}
```

To actually select some pages, we need to intercept the navigation menu clicks. We'll be using the code we wrote earlier to capture the event and prevent the browser from navigating to the link:

javascripts/ch4/07-swap.js *(excerpt)*

```
$("#tab-bar a").bind('click', function(e) {
  e.preventDefault();
  // Swap pages!
});
```

And here's the trick: the links point to our page elements by using the anchor syntax of a hash symbol (`#`), followed by a fragment identifier. It coincidently happens that jQuery uses that exact same syntax to select elements by `id`, so we can funnel the `hash` property of the `click` event directly into jQuery to select the destination page. Very sneaky:

javascripts/ch4/07-swap.js *(excerpt)*

```
$("#tab-bar a").bind('click', function(e) {
  e.preventDefault();
  var nextPage = $(e.target.hash);
  $("#pages .current").removeClass("current");
  nextPage.addClass("current");
});
```

With the target page acquired, we can hide the current page by removing the `current class` and passing it to the destination page. Swapping between pages now works as expected, but there's a slight problem: the selected icon in the tab bar fails to change when you navigate to another page. Looking back at our CSS, you'll remember that the tab bar's appearance is due to a `class` set on the containing `ul` element; it's a `class` that's the same as the current page `div` element's `id`. So all we need to do is slice out the hash symbol from our string (using `slice(1)` to remove the first character), and set that as the `ul`'s `class`:

javascripts/ch4/07-swap.js *(excerpt)*

```
$("#tab-bar a").bind('click', function(e) {
  e.preventDefault();
  var nextPage = $(e.target.hash);
  $("#pages .current").removeClass("current");
  nextPage.addClass("current");
  $("#tab-bar").attr("className", e.target.hash.slice(1));
});
```

Fading with WebKit Animations

The page swap we just implemented is as straightforward as it gets. This has its advantages—it stays out of our users' way, for one. That said, well-placed transitions between pages not only make your apps sexier, they can provide a clear visual cue to the user as to where they're being taken.

After the original iPhone was released, web developers leapt to re-implement the native transition effects in JavaScript, but the results were less than ideal, often containing lags and jumps that were very noticeable and distracting to users. The solution largely was to ditch JavaScript for moving large DOM elements, and instead turn to the new and hardware-accelerated CSS3 transitions and animations.

Before we worry about the transitions, though, we need to lay some groundwork. To fling DOM elements around, we need to be able to show, hide, and position them at will:

stylesheets/transitions.css *(excerpt)*

```
#pages {
  position: relative;
}
#pages > div {
  display:none;
  position: absolute;
  top: 0;
  left: 0;
  width: 100%;
}
```

By positioning the elements absolutely, we've moved every page up into the top-left corner, giving us a neat stack of invisible cards that we can now shuffle around and animate. They're not all invisible, though; remember that in our HTML, we gave our default page the `class` of `current`, which sets its `display` property to `block`.

The difference this time is that we're going to apply CSS animations to the pages. The incoming (new) page, and the outgoing (current) page will have equal but opposite forces applied to them to create a smooth-looking effect. There are three steps required to do this:

1. Set up the CSS animations.
2. Trigger the animation by setting the appropriate classes on the pages.
3. Remove the non-required classes when the animation is finished, and return to a non-animating state.

Let's start on the CSS. There are many approaches you can take with the problem of emulating native page transitions. We'll adopt a flexible method that's adapted from the jQTouch library. This

is a modular approach, where we control transitions by applying and removing the relevant parts of an animation to each page.

Before we dive into that, though, a quick primer on CSS3 animations. These are currently supported only in WebKit browsers with `-webkit-` vendor prefixes. A CSS3 animation is made up of a series of keyframes grouped together as a named animation, created using the `@-webkit-keyframes` rule. Then we apply that animation to an element using the `-webkit-animation-name` property. We can also control the duration and easing of the animation with the `-webkit-animation-duration` and `-webkit-animation-timing-function` properties, respectively. If you're new to animations, this is probably sounding more than a little confusing to you right now; never mind, once you see it in practice, it'll be much clearer.

So let's apply some animations to our elements. First up, we'll set a timing function and a duration for our animations. These dictate how long a transition will take, and how the pages are eased from the start to the end point:

stylesheets/transitions.css *(excerpt)*

```
.in, .out {
  -webkit-animation-timing-function: ease-in-out;
  -webkit-animation-duration: 300ms;
}
```

We've placed these properties in generic classes, so that we can reuse them on any future animations we create.

Next, we need to create our keyframes. To start with, let's simply fade the new page in:

stylesheets/transitions.css *(excerpt)*

```
@-webkit-keyframes fade-in {
  from { opacity: 0; }
  to { opacity: 1; }
}
```

In the above rule, `fade-in` is the name of the animation, which we'll refer to whenever we want to animate an element using these keyframes. The `from` and `to` keywords allow us to declare the start and end points of the animation, and they can include any number of CSS properties you'd like to animate. If you want more keyframes in between the start and end, you can declare them with percentages, like this:

stylesheets/transitions.css *(excerpt)*

```
@-webkit-keyframes fade-in-out {
  from { opacity: 0; }
  50% { opacity: 1; }
  to { opacity: 0; }
}
```

With our keyframes declared, we can combine them with the previous direction classes to create the final effect. For our fade, we'll use the animation we defined above, and also flip the `z-index` on the pages to make sure the correct page is in front:

stylesheets/transitions.css *(excerpt)*

```
.fade.in {
  -webkit-animation-name: fade-in;
  z-index: 10;
}
.fade.out {
  z-index: 0;
}
```

By declaring `-webkit-animation-name`, we're telling the browser that as soon as an element matches this selector, it should begin the named animation.

With this CSS in place, we can move to step two. We'll start by applying our animation to a single navigation item, then broaden it out later so that it will work for our whole tab bar.

The page we're fading to (`#sightings`) will need to have three different classes added to it: `current` to make the page visible, `fade` to add our animation, and `in` to apply our timing function and duration. The page we're fading from (`#spots`) is visible, so it will already have the `current` `class`; we only need to add the `fade` and `out` classes:

```
var fromPage = $("#spots"),
    toPage = $("#sightings");

$("#tab-sighting a").click(function(){
  toPage.addClass("current fade in");
  fromPage.addClass("fade out");
});
```

This gives us a nice fading effect when we click on the "Add a sighting" tab, but now the pages are stuck—stacked atop one another. This is because those `class` names are still there, so the pages now have `current` and they're both visible. Time to remove them! We'll do this by binding to the `webkitAnimationEnd` event, which fires when the transition is complete. When this event fires, we can remove all three classes from the original page, and the `fade` and `in` classes from the new page.

Additionally, we must remember to unbind the `webkitAnimationEnd` event so that we don't go adding on extra handlers the next time we fade from the page:

```
var fromPage = $("#spots"),
  toPage = $("#sightings");

$("#tab-sighting a").click(function(){
  toPage
    .addClass("current fade in")
    .bind("webkitAnimationEnd", function(){
      // More to do, once the animation is done.
      fromPage.removeClass("current fade out");
      toPage
        .removeClass("fade in")
        .unbind("webkitAnimationEnd");
    });
  fromPage.addClass("fade out");
});
```

There we go. Our page is now fading nicely; however, there are a few problems with our code. The first is structural. It will become quite ugly if we have to replicate this same click handler for each set of pages we want to transition to! To remedy this, we'll make a function called `transition()` that will accept a page selector and fade from the current page to the new one provided.

While we're at it, we can replace our `bind()` and `unbind()` calls with jQuery's `one()` method. This method will accomplish the same task—it binds an event, and then unbinds it the first time it's fired—but it looks a lot cleaner:

javascripts/ch4/08-fade.js *(excerpt)*

```
function transition(toPage) {
  var toPage = $(toPage),
    fromPage = $("#pages .current");

  toPage
    .addClass("current fade in")
    .one("webkitAnimationEnd", function(){
      fromPage.removeClass("current fade out");
      toPage.removeClass("fade in")
    });
  fromPage.addClass("fade out");
}
```

Generalizing Functions

You might spy that we've hardcoded the current page selector inside our function. This makes our code smaller, but reduces the reusability of the function. If you are building a larger framework intended for more general use, you'd probably want to accept the `fromPage` as a parameter, too.

Great. Now we have a reusable function that we can employ to fade between any of the pages in our app. We can pull the link targets out of the tab bar the same way we did earlier, and suddenly every page swap is a beautiful fade:

javascripts/ch4/08-fade.js *(excerpt)*

```
$("#tab-bar a").click(function(e) {
  e.preventDefault();
  var nextPage = $(e.target.hash);
  transition(nextPage);
  $("#tab-bar").attr("className", e.target.hash.slice(1));
});
```

There's still a major problem, though, and it's one you'll notice if you try to test this code on a browser that lacks support for animations, such as Firefox. Because we're relying on the `webkitAnimationEnd` event to remove the `current` `class` from the old page, browsers that don't support animations—and therefore never fire that event—will never hide the original page.

Browser Testing

This bug—which would render the application completely unusable on non-WebKit browsers—highlights the importance of testing your code on as many browsers as possible. While it can be easy to assume that every mobile browser contains an up-to-date version of WebKit (especially if you own an iPhone or Android), the real mobile landscape is far more varied.

This problem is easy enough to solve. At the end of our `transition()` function, we'll drop in some feature detection code that will handle the simplified page swap in the absence of animations:

javascripts/ch4/08-fade.js *(excerpt)*

```
function transition(toPage) {
  ⋮
  // For non-animatey browsers
  if(!("WebKitTransitionEvent" in window)){
    toPage.addClass("current");
    fromPage.removeClass("current");
    return;
  }
}
```

With this code in place, our app now produces our beautiful fade transition on WebKit browsers, but still swaps pages out effectively on other browsers.

There's still one slightly buggy behavior, one you might notice if you become a little excited and start clicking like crazy. If you click on the link to the current page—or if you tap quickly to start an animation when the previous one has yet to complete—the `class` attributes we're using to manage the application's state will be left in an inconsistent state. Eventually, we'll end up with *no* pages with the `current class`—at which point we'll be staring at a blank screen.

It's relatively easy to protect against these cases. We just need to ensure our `toPage` is different from our `fromPage`, and that it doesn't already have the `current class` on it. This safeguard goes after the variable declaration, and before any class manipulations:

javascripts/ch4/08-fade.js *(excerpt)*

```
function transition(toPage) {
  var toPage = $(toPage),
    fromPage = $("#pages .current");

  if(toPage.hasClass("current") || toPage === fromPage) {
    return;
  };
  ⋮
```

Sliding

Awesome work—our first animated transition is up and running! Let's move on to the next one: an extremely common mobile interaction for master-detail pages, such as our Spots and Stars listings. On mobile devices, it's common for the detail page to slide in from the right side of the screen, as if it had been hiding there the whole time. When the user returns to the master page, the transition works in reverse, providing the user with a clear visual model of the app's information hierarchy.

Creating a slide or "push" transition is very easy now that we've learned to fade—we just need to update our CSS animations. To perform a push, we need to animate the current page off the screen to the left while simultaneously moving the new page in from the right. It's a bit trickier than just fading, but only just.

First, we'll create an animation for moving the current screen away. We'll use the `-webkit-transform` property to animate the `div` element's horizontal location via the `translateX` command.

CSS Transforms

If you're unfamiliar with CSS3 transforms, don't worry. They're simply an easy way to act on the position and shape of an element as its rendered. In addition to the translateX transform we're using here, you also have access to translateY (unsurprisingly), translate (for both X- and Y-axis translation), as well as `rotate`, `skew`, and `scale` functions. For the full list and examples of how they can be used, check out the W3C's CSS 2D Transforms Module.[4]

The screen starts at X=0 and then moves out of sight by translating to -100%. This is a nifty trick. If we translate by a negative amount, the screen moves to the left. By moving 100% to the left, the screen is gone. Once we've defined the keyframes, we can then assign the animation to the `.push.out` selector (remember that our existing `out class` provides a default duration and timing function for our animations):

stylesheets/transitions.css *(excerpt)*

```
/* Screen pushes out to left */
@-webkit-keyframes outToLeft {
  from { -webkit-transform: translateX(0); }
  to { -webkit-transform: translateX(-100%); }
}
.push.out {
  -webkit-animation-name: outToLeft;
}
```

Similarly, we'll define an animation for the new screen to fly in from the right, taking the place of the old one:

stylesheets/transitions.css *(excerpt)*

```
/* Screen pushes in from the right */
@-webkit-keyframes inFromRight {
  from { -webkit-transform: translateX(100%); }
  to { -webkit-transform: translateX(0); }
}
.push.in {
  -webkit-animation-name: inFromRight;
}
```

For the JavaScript, we could recycle the transition function we made for the fade, calling it `transition_push()`, for example. We'd then just need to change all the instances of `fade` to `push`. And then again for `flip` when we want to implement flip transitions, and so on. On second thoughts, it would be nicer to pass the transition type in as a parameter to our `transition()` function:

[4] http://www.w3.org/TR/css3-2d-transforms/#transform-functions

javascripts/ch4/09-slide.js *(excerpt)*

```
function transition(toPage, type) {
  ⋮
  toPage
    .addClass("current " + type + " in")
    .one("webkitAnimationEnd", function(){
      fromPage.removeClass("current " + type + " out");
      toPage.removeClass(type + " in");
    });
  fromPage.addClass(type + " out");
}
```

Now when we create CSS animations for a new transition, they'll automatically be available to use in our script. We just pass the new name in:

```
transition(nextPage, "push");
```

We'd like the push transition to be used for navigating down from a list to a detail page, so we need to add some new click handlers for the list items:

javascripts/ch4/09-slide.js *(excerpt)*

```
$("#spots-list li").click(function(e){
  e.preventDefault();
  transition("#page-spot", "push");
});
$("#stars-list li").click(function(e){
  e.preventDefault();
  transition("#page-star", "push");
});
```

With this, the detail pages will slide in from the right to replace the list pages. As we outlined earlier, though, what we'd also like is for the reverse to occur when the user is navigating back *up* the app's hierarchy. Next up, we'll look at building out that functionality, and, while we're at it, adding support for going "back."

Going Backwards

The user is now looking at a page of details about some crazy celebrity, and they're bored. They want a new crazy celebrity to read about, so they go looking for the Back button. But going back is more than just swapping the source and destination pages again, because the animations we applied need to be reversed: the old page needs to slide back from the left into view.

But that's getting ahead of ourselves; first, we need a Back button. We've provided one up in the header of each page in the form of an `a` element that's styled to look all button-like:

ch4/10-back.html *(excerpt)*

```
<div class="header">
  <h1>Spots</h1>
  <a href="#" class="back">Back</a>
</div>
```

And of course, we must have a handler to perform an action when the button is clicked:

javascripts/ch4/10-back.js *(excerpt)*

```
$("#spot-details .back").click(function(){
  // Do something when clicked …
});
```

Next, we need to recreate all our CSS animations—but in reverse. We've already created `inFromRight` and `outFromLeft` animations; we need to add two more to complement them: `inFromLeft` and `outToRight`. Once these are defined, they have to be attached to our elements with CSS selectors. We'll continue the modular approach, and use a combination of class selectors to leverage our existing properties:

stylesheets/transitions.css *(excerpt)*

```
@-webkit-keyframes inFromLeft {
  from { -webkit-transform: translateX(-100%); }
  to { -webkit-transform: translateX(0); }
}
.push.in.reverse {
  -webkit-animation-name: inFromLeft;
}
@-webkit-keyframes outToRight {
  from { -webkit-transform: translateX(0); }
  to { -webkit-transform: translateX(100%); }
}
.push.out.reverse {
  -webkit-animation-name: outToRight;
}
```

The next step is to work the new `class` into our `transition()` function. We'll add a third parameter, *`reverse`*, that accepts a Boolean value. If the value is `false`, or if it's not provided at all, we'll do the forward version of the transition. If the value is `true`, we'll append the `reverse` `class` to all the class manipulation operations:

javascripts/ch4/10-back.js *(excerpt)*

```
function transition(toPage, type, reverse){
  var toPage = $(toPage),
    fromPage = $("#pages .current"),
    reverse = reverse ? "reverse" : "";

  if(toPage.hasClass("current") || toPage === fromPage) {
    return;
  };
  toPage
    .addClass("current " + type + " in " + reverse)
    .one("webkitAnimationEnd", function(){
      fromPage.removeClass("current " + type + " out " + reverse);
      toPage.removeClass(type + " in " + reverse);
    });
  fromPage.addClass(type + " out " + reverse);
}
```

If we pass in `true` now, the new page will be assigned the `class` attribute `push in reverse`, and the old page will be assigned `push out reverse`—which will trigger our new backwards animations. To see it in action, we'll add a call to `transition()` in our Back button hander:

javascripts/ch4/10-back.js *(excerpt)*

```
$("#page-spot .back").click(function(e){
  e.preventDefault();
  transition("#page-spots", "push", true);
});
```

Managing History

The Back button works, but it's a bit "manual" at the moment. For every page in our app, we'd have to hook up a separate handler to go back. Worse still, some pages could be reached via a number of different routes, yet our current solution only goes back to a fixed page. To combat these problems, we'll create our very own history system that will keep track of each page users visit, so that when they hit the Back button, we know where we should send them.

To start with, we'll create a `visits` object, which will contain a `history` array and some methods to manage it:

javascripts/ch4/11-history.js *(excerpt)*

```
var visits = {
  history: [],
  add: function(page) {
```

```
    this.history.push(page);
  }
};
```

Our `visits` object will maintain a stack of visited pages in the `history` array. The `add()` method takes a page and prepends it to the stack (via the JavaScript `push()` function, which adds an element to the end of an array). We'll call this method from inside our `transition()` function, so that every page will be added before it's shown:

javascripts/ch4/11-history.js *(excerpt)*

```
function transition(toPage, type, reverse) {
  var toPage = $(toPage),
    fromPage = $("#pages .current"),
    reverse = reverse ? "reverse" : "";

  visits.add(toPage);
  ⋮
}
```

Centralizing Page Changes

The assumption that every transition corresponds to a page change is convenient for us, otherwise we'd have to call `visits.add()` everywhere we do a transition. However, there might be times when you want to do a transition to a new page, but not include it as a page change—for example, if you have some kind of slide-up dialog. In this case, you could create a `changePage()` function that handles both history management and transitioning. We'll be doing this in the next section.

The next item to think about is our Back button. We only want it to be shown if there's a history item to revert to. We'll add a helper method to the `visits` object to check for us. Because the first page in the history will be the current page, we need to check that there are at least two pages:

javascripts/ch4/11-history.js *(excerpt)*

```
var visits = {
  ⋮
  hasBack: function() {
    return this.history.length > 1;
  }
}
```

Now that we have this helper, we can use it in our transition code to show or hide the Back button accordingly. The `toggle()` jQuery function is very useful here; it accepts a Boolean value, and either shows or hides the element based on that value:

javascripts/ch4/11-history.js *(excerpt)*

```
function transition(toPage, type, reverse) {
  var toPage = $(toPage),
    fromPage = $("#pages .current");
    reverse = reverse ? "reverse" : "";

  visits.add(toPage);
  toPage.find(".back").toggle(visits.hasBack());
  ⋮
```

Good! Now we need some logic in our `visits` object to handle a back event. If there is history, we'll pop the first item (the current page) off the top of the stack. We don't actually need this page—but we have to remove it to reach the next item. This item is the previous page, and it's the one we return:

javascripts/ch4/11-history.js *(excerpt)*

```
var visits = {
  ⋮
  back: function() {
    if(!this.hasBack()){
      return;
    }
    var curPage = this.history.pop();
    return this.history.pop();
  }
}
```

Push and Pop

The `push()` and `pop()` methods add or remove an element from the end of an array, respectively. Both methods modify the original array in place. The `pop()` method returns the element that has been removed (in our example, we use this to get the previous page), whereas the `push()` method returns the new length of the array.

Finally, we can wire up all our application's Back buttons. When a request to go back is issued, we grab the previous page and, if it exists, we transition back to it. We just replace our hardcoded click handler with a general-purpose one:

javascripts/ch4/11-history.js *(excerpt)*

```
$(".back").live("click",function(){
  var lastPage = visits.back();
  if(lastPage) {
    transition(lastPage, "push", true);
  }
});
```

There's still a problem, though: we never add the *initial* page to the history stack, so there's no way to navigate back to it. That's easy enough to fix—we'll just remove the `current class` from the initial `div`, and call our transition function to show the first page when the document loads:

javascripts/ch4/11-history.js *(excerpt)*

```
$(document).ready(function() {
  ⋮
  transition($("#page-spots"), "show");
});
```

To hook up that "show" transition, we'll reuse our fade animation, but with an extremely short duration:

stylesheets/transitions.css *(excerpt)*

```
.show.in {
  -webkit-animation-name: fade-in;
  -webkit-animation-duration: 10ms;
}
```

Many native apps only track history between master and details pages; in our case, for example, a list of stars leads to the star's details, and the Back button allows you to jump back up to the list. If you change areas of the application (for example, by clicking on one of the main navigation links), the history is reset. We can mimic this behavior by adding a `clear()` method:

javascripts/ch4/11-history.js *(excerpt)*

```
var visits = {
  ⋮
  clear: function() {
    this.history = [];
  }
}
```

This simply erases our history stack. We'll call this method whenever the user moves to a new section:

javascripts/ch4/11-history.js *(excerpt)*

```
$("#tab-bar a").click(function(e){
  // Clear visit history
  visits.clear();
  ⋮
});
```

This has a very "app" feeling, and, as an added bonus, we don't have to wire up so many Back button events!

Back with Hardware Buttons

Our current Back button system is good, but it doesn't take into account the fact that a mobile device will often have its own Back button—either in the form of a physical button, or a soft button in the browser. As it stands, if a user hits their device's Back button after clicking a few internal links in our app, the browser will simply move to the last HTML page it loaded, or exit completely. This will definitely break our users' illusion of our site as a full-fledged app, so let's see if we can find a fix for this problem.

What we really need is to be able to listen to, and modify, the browser's built-in history, instead of our own custom stack of pages. To accomplish this, the HTML5 History API[5] is here to help us out.

The History API lets us add pages to the history stack, as well as move forward and backwards between pages in the stack. To add pages, we use the `window.history.pushState()` method. This method is analogous to our `visits.add()` method from earlier, but takes three parameters: any arbitrary data we want to remember about the page; a page title (if applicable); and the URL of the page.

We're going to create a method `changePage()` that combines both adding a page using the history API, and doing our regular transition. We'll keep track of the transition inside the history, so that when the user presses back, we can look at the transition and do the opposite. This is nicer than our previous version, where we'd only ever do a reverse slide for the back transition.

Here's a first stab at writing out this new method:

javascripts/ch4/12-hardware-back.js *(excerpt)*

```
function changePage(page, type, reverse) {

  window.history.pushState({
    page: page,
    transition: type,
    reverse: !!reverse
  }, "", page);

  // Do the real transition
  transition(page, type, reverse)
}
```

[5] http://www.w3.org/TR/html5/history.html

The first parameter to `pushState()` is referred to as the **state object**. You can use it to pass any amount of data between pages in your app in the form of a JavaScript object. In our case, we're passing the page, the transition type, and whether or not it's a reverse transition.

To use this new function in our code, we merely change all occurrences of `transition()` to `changePage()`, for example:

```
changePage("#page-spots", "show");
```

Now, as the user moves through our application, the history is being stored away. If they hit the physical Back button, you can see the page history in the URL bar, but nothing special happens. This is to be expected: we've just pushed a series of page strings onto the history stack, but we haven't told the app how to navigate back to them.

The `window.onPopState` event is fired whenever a real page load event happens, or when the user hits Back or Forward. The event is fed an object called *`state`* that contains the state object we put there with `pushStack()` (if the state is undefined, it means the event was fired from a page load, rather than a history change—so it's of no concern). Let's create a handler for this event:

javascripts/ch4/12-hardware-back.js *(excerpt)*

```
window.addEventListener("popstate", function(event) {
  if(!event.state){
    return;
  }

  // Transition back - but in reverse.
  transition(
    event.state.page,
    event.state.transition,
    !event.state.reverse
  );
}, false);
```

Where's jQuery?

For this example, we've just used a standard DOM event listener rather than the jQuery `bind()` method. This is just for clarity for the `popstate` event. If we bound it using `$(window).bind("popstate", …)`, the event object passed to the callback would be a jQuery event object, not the browser's native `popstate` event. Usually that's what we want, but jQuery's event wrapper doesn't include the properties from the History API, so we'd need to call `event.originalEvent` to retrieve the browser event. There's nothing wrong with that—you can feel free to use whichever approach you find simplest.

Fantastic! The animations all appear to be working in reverse when we hit the browser Back button … or are they? If you look closely, you might notice something strange. Sometimes we see "slide" transitions that should be simple "show" transitions, and vice versa. What's going on?

Actually, we have an off-by-one error happening here: when moving backwards, we don't want to use the transition of the page we are transitioning *to*, but the page we are transitioning *from*. Unfortunately, this means we need to call `pushState()` with the *next* transition that happens. But we're unable to see the future … how can we know what transition is going to happen next?

Thankfully, the History API provides us with another method, `replaceState()`. It's almost identical to `pushState()`, but instead of adding to the stack, it *replaces* the current (topmost) page on the stack. To solve our problem, we'll hang on to the details of the previous `pushState()`; then, before we add the next item, we'll use `replaceState()` to update the page with the "next" transition:

javascripts/ch4/12-hardware-back.js *(excerpt)*

```
var pageState = {};
function changePage(page, type, reverse) {
  // Store the transition with the state
  if(pageState.url){
    // Update the previous transition to be the NEXT transition
    pageState.state.transition = type;
    window.history.replaceState(
      pageState.state,
      pageState.title,
      pageState.url);
  }
  // Keep the state details for next time!
  pageState = {
    state: {
      page: page,
      transition: type,
      reverse: reverse
    },
    title: "",
    url: page
  }
  window.history.pushState(pageState.state, pageState.title, pageState.url);

  // Do the real transition
  transition(page, type, reverse)
}
```

We also need to update our `pageState` variable when the user goes back; otherwise, it would fall out of sync with the browser's history, and our `replaceState()` calls would end up inserting bogus entries into the history:

javascripts/ch4/12-hardware-back.js *(excerpt)*

```
window.addEventListener("popstate", function(event) {
  if(!event.state){
    return;
  }
  // Transition back - but in reverse.
  transition(
    event.state.page,
    event.state.transition,
    !event.state.reverse
  );
  pageState = {
    state: {
      page: event.state.page,
      transition: event.state.transition,
      reverse: event.state.reverse
    },
    title: "",
    url: event.state.page
  }
}, false);
```

There we go. The physical Back button now works beautifully. But what about our custom application Back button? We can wire that up to trigger a history event, and therefore tie into all that History API jazz we just wrote using a quick call to `history.back()`:

javascripts/ch4/12-hardware-back.js *(excerpt)*

```
$(".back").live("click",function(e){
  window.history.back();
});
```

Now our application Back button works exactly like the browser or physical Back button. You can also wire up a Forward button and trigger it with `history.forward()`, or skip to a particular page in the stack with `history.go(-3)`. You might have noticed that we've been a bit quiet on the Forward button handling. There are two reasons for this: first, most mobile browsers lack a Forward button, and second, it's impossible to know if the `popstate` event occurred because of the Back or the Forward button.

The only way you could get around this pickle would be to combine the `popstate` method with the manual history management system we built in the previous section, looking at the URLs or other data to determine the direction of the stack movement. This is a lot of work for very little return in terms of usability, so we'll settle for the history and back functionality we've built, and move on to the next challenge.

Ajax

We've now learned how to transition between pages smoothly and without reloading, but so far we've only done this with static content. We need to be able to load our pages dynamically, and *then* transition to them.

The good news is that there's comparatively excellent support for Ajax on high-end mobile devices—so Ajax work remains largely the same as you're used to on the desktop. Of course, you have to consider that the average data connection will be an order of magnitude slower (and more expensive), so it's best to keep your bandwidth use to an absolute minimum.

There are two approaches we could take to loading dynamic data into our application. We could load in the raw data (such as a list of JSON objects representing recently spotted celebrities) and merge it into the application's HTML—creating list elements and appending them to the page. Or, we could load the entire HTML contents straight from the server and dump it directly into our page. The latter approach sounds more straightforward—so let's try that first to become familiar with Ajax in the mobile environment. After that, we'll take a look at handling other data formats.

Fetching HTML

The first thing we'll need if we want to retrieve data from a server is … a server! If you try and grab data from a file:// protocol URL (which will be the case if you're testing pages by double-clicking an **index.html** file), you'll hit the dreaded "Access-Control-Allow-Origin" error.

Servers

As this is a book about building mobile web apps in HTML, CSS, and JavaScript, covering the details of setting up a server to deliver your data is unfortunately beyond our scope. We'll spend the rest of this chapter looking at examples of Ajax functionality we can add to Startrackr, but if you want to try these examples for yourself, you'll need to set up a server to deliver the appropriate data.

Assuming we have a server—be it running on our machine locally, on a VM, or on the Internet—we now need somewhere to dump the HTML chunks when they're returned from the server. For this we only require a skeleton of the page we've been working with, to act as a container that we fill. Because we'll be returning the same markup as before, our original CSS will apply to the contents without needing to be modified. Here's that skeleton:

ch4/13-ajax.html *(excerpt)*

```
<div id="pages">
  <div id="page-spots" class="page-spots"></div>
  <div id="page-spot" class="page-spots"></div>
  <div id="page-sightings" class="page-sightings"></div>
```

```
  <div id="page-stars" class="page-stars"></div>
  <div id="page-star" class="page-stars"></div>
</div>
```

All the content for each of those sections is back in the **spots.html**, **new.html**, and **stars.html** files—just like a regular website, except that now we'll be using Ajax to load that content into this empty skeleton.

With this basic HTML in place, the next step is to stop the link from being followed when clicked, as we did with our transition function earlier—with `preventDefault()`. Then we can execute our Ajax. The jQuery `load()` function is perfect for our needs: it loads HTML from a URL, and provides a mechanism for choosing which part of the document to return. This is great, because we have no need for the whole page, with the `head` and `meta` tags—we only want the contents of the `body`. Using `load()` means we don't need a special version of our HTML page for Ajax, and any updates will only have to be made in one place.

To accomplish this, we use the `load()` function with a string parameter consisting of the URL we want, followed by a space, followed by a jQuery selector string. The contents of the element matched by that selector will be inserted into the element from which `load()` was called. The content we want to insert is contained in the `.wrapper div` of the target page. So, when the Spots link is clicked, we want to call `load()` on our `#spots` container and pass in the string `"spots.html .wrapper"`:

javascripts/ch4/13-ajax.js *(excerpt)*

```
$("#tab-spots a").click(function(e){
  e.preventDefault();
  $("#page-spots").load("spots.html .wrapper");
});
```

Loading HTML Snippets

If you're unfamiliar with jQuery, you might be wondering how it loads in a small section of HTML via Ajax. There's no magic here; it actually loads the entire page and dumps it into a `div` element that exists outside the DOM. The filtering is done on this element, and the results are inserted in the correct position. Very handy, although of course it means transmitting more data over the network than you actually end up using. For most real-world applications, you'll probably want to pull data in XML or JSON format and insert it into your HTML on the client side. We'll be looking at how this can be done shortly. For now, though, we'll stick with using `load()` to keep it simple, and focus on demonstrating how the various Ajax methods work, as well as how they're best used in the context of a mobile app.

This will load the relevant HTML into the right container, but there are a few more tasks that need attention—most notably, making the new content visible! Fortunately, `load()` allows you to specify

a callback function that will be executed once the Ajax call has completed. Inside this callback, we'll transition the page into view:

javascripts/ch4/13-ajax.js *(excerpt)*

```
$("#tab-spots a").click(function(e){
  e.preventDefault();
  $("#page-spots").load("spots.html .wrapper", function() {
    transition('#page-spots', "fade", false);
  });
});
```

Ajaxifying Links

Adding events to each navigation item is a tedious way to wire up our site. We made an Ajax loader for the Spots page just now, but we'd have to duplicate this code multiple times for it to work for all the links. Instead, we can take advantage of our site's consistent structure and concoct a system to do it programmatically, based on the contents of the navigation elements. Doing this gives us a level of progressive enhancement: our links are designed to work as normal, but our system intercepts them when clicked and loads the content via Ajax. This method is sometimes known as **Hijax**.

We'll then generalize the Ajax code we wrote so that it can be applied to any link we pass it. There are two pieces of data needed for that: the URL to load, and the name of the container to dump it in. This is where having conventions is important. As long as our pages and classes are named in a consistent manner, we can easily create code that targets all our links:

javascripts/ch4/14-hijax.js *(excerpt)*

```
function loadPage(url, pageName) {
  $("#" + pageName).load(url + " .wrapper", function(){
    console.log(this);
    transition("#" + pageName, "fade", false);
  });
};
```

This function is almost identical to the code from before, except we've replaced the page names and URLs with variables. Now we can load a page programmatically, for example:

```
loadPage("spots.html", "spots");
```

If fact, we need to load a page by default when the application loads, so we can place that line of code inside the `document.ready` handler. This will load up the Spots page via Ajax as our home page.

Data Caching

Because the `load()` method pulls in an entire HTML file, the results can be cached by the browser, which means that changes you make in the page being loaded may not be reflected right away. You can disable the cache globally (for all Ajax requests) with `$.ajaxSetup({ cache: false });`, or, if it's just for one particular call, you can append a timestamp to the URL so that each request will be seen by the browser as different. So, instead of loading `url + " #wrapper"`, you can load `url + "?" + new Date().getTime() + " #wrapper"`.

Finally, we want to call our function whenever any navigation items are clicked. Remember, we have to extract two pieces of data to pass to our function: the URL and the page name. The URL is simple—it's the `href` value of the link itself. For the page name, there are many approaches we could take: we could give our containers the same names as the files (minus the **.html**), or we could add some extra data into the link itself. The latter has become easier with the addition of custom data attributes in HTML5, which let us annotate elements with key/value pairs:

ch4/14-hijax.html *(excerpt)*

```
<ul id="tab-bar">
  <li>
    <a data-load="spots" href="spots.html">Spots</a>
  </li>
  <li>
    <a data-load="sightings" href="new.html">Add a sighting</a>
  </li>
  <li>
    <a data-load="stars" href="stars.html">Stars</a>
  </li>
</ul>
```

A data attribute starts with `data-` and is followed by your key name. You can then provide it with whatever value you like. According to the spec, these values should be retrieved using the element's `dataset` property; however, this is yet to be widely supported, so your best bet is to use the standard `getAttribute()` function (as in, `myElement.getAttribute("data-load")`), or, if you're using jQuery, the `attr()` method:

javascripts/ch4/14-hijax.js *(excerpt)*

```
$("#tab-bar a").click(function(e){
  e.preventDefault();
  var url = e.target.href;
  var pageName = $(this).attr("data-load");
  loadPage(url, pageName);
});
```

Et voilà! Any link inside the `#tab-bar` element will fire our `loadPage()` function, and we'll transition from the current page to the new page. You can easily extend this system to also specify the transition type, if you like.

One current problem is that if the page takes a long time to load, the user has no idea what's going on (and they'll probably become click-happy and try to load another page). The obvious solution is a "loading" indicator; jump over to Chapter 6 if you're keen to add one now.

Templating

Pulling in an entire dollop of ready-to-go HTML via Ajax makes it nice and easy to stick in your page, but typically this isn't the data format you'll receive from a web service or API. This means it's our responsibility to turn data into markup. Even if you're in control of the back end, it still makes sense to transmit your site data in XML or JSON rather than HTML, since those files will usually be much smaller, and require less of your mobile users' precious bandwidth.

So, how will this work? We start by receiving a list of data from the server. For each item in the list, we want to create an HTML fragment (such as a list item) and inject some values from the data. Finally, we want to add each fragment into the correct place in the page. There are a few common ways to approach this situation:

- build the HTML fragments ourselves, in JavaScript
- duplicate a "template" DOM element and change the contents of the relevant tags
- use a templating engine

We'll quickly visit each approach and see what works best in which contexts. We'll be assuming our data source is in the JSON format, and looks a little like this:

data/spots.json *(excerpt)*

```
[{
  "id": 4,
  "name": "Planet Bollywood",
  "distance": "400m",
  "sightings": 10
}, {
  "id": 7,
  "name": "Soft Rock Café",
  "distance": "1.1Km",
  "sightings": 3
}]
```

It's a simple array, with each element consisting of an object containing details about locations. This is a very common (albeit simplified) format for the data that we're likely to receive. To fetch it into our application, we can use jQuery's `getJSON()` method:

javascripts/ch4/15-templating.js *(excerpt)*

```
$.getJSON("../data/spots.json", function(data){
  // Got JSON, now template it!
});
```

With some data in our pocket, it's time to show it to the world. The first of our three approaches is to loop over the `data` array, building up HTML strings that can then be added into the DOM:

javascripts/ch4/15-templating.js *(excerpt)*

```
$.getJSON("../data/spots.js", function(data){
  // Got JSON, now template it!
  var html = "";
  for(var i = 0; i < data.length; i++) {
    html += "<li><a href='#'>";
    html += "<h2>" + data[i].name + "</h2>";
    html += "</a></li>";
  }
  $("#spots-list").append(html);
});
```

This is a very old-school approach, and while it can work for very simple examples, it tends to scale poorly: it's error-prone, difficult to maintain, and degenerates fairly quickly into a huge mess of mashed-together HTML and data.

The separation of data and presentation is important, and this is especially true for web applications. Using the previous method, if a designer wanted to make changes to the HTML, they'd either have to know some JavaScript (and potentially risk breaking something), or come and bother you about it. We would prefer to avoid being bothered, so a better solution is to include the HTML where it belongs—in the HTML page:

ch4/16-templating-2.html *(excerpt)*

```
<div id="tmpl-simple" style="display:none;">
  <li>
    <a href="spot.html" data-load="spot">
      <h2></h2>
      <span class="relative-distance"></span>
      <span class="sightings"></span>
    </a>
  </li>
</div>
```

We've constructed a generic fragment that's empty of any data. When we receive data from the server, we can clone this template and fill it with our data. It can be as complex as it needs to be, as long as there are some hooks for us to inject our data into. Of course, you don't want an empty row in the list shown to the user—so the template needs to be hidden. In the above example, this is done using `display: none;`.

To make use of the fragment, we'll clone it with jQuery's `clone()` method, and then inject all the pieces of data into their correct elements using the `text()` method:

javascripts/ch4/16-templating-2.js *(excerpt)*

```
$.getJSON("../data/spots.json", function(data){
  // Got JSON, now template it!
  $.each(data, function(){
    var newItem = $("#tmpl-simple").clone();

    // Now fill in the fields with the data
    newItem.find("h2").text(this.name);
    newItem.find(".relative-distance").text(this.distance);
    newItem.find(".sightings").text(this.sightings);

    // And add the new list item to the page
    newItem.children().appendTo("#spots-list")
  });
  transition("#spots", "show");
});
```

This solution works well, leaving all the HTML in one file and all the JavaScript in another. It's much nicer than our first approach, and is suitable for small static templates. But it's less ideal if the HTML is continually changing, and for large pages it requires us to write a lot of generic JavaScript to replace all the field values.

This is the kind of mind-numbing code that we want to automate away. Perhaps we could add some tokens in the HTML that could be automatically replaced from a given data object? That's quite a good idea, and it turns out plenty of developers have had it before: it's called a **templating engine**. Choosing a templating engine is like choosing a text editor: there are a lot out there, they have a many different features, most of them are fine, and ultimately it's up to you which one you (or your company) likes the best.

One "big one" that you'll see mentioned time and time again is Mustache.[6] It's available for many programming languages, and the template format is the same for all of them—which makes it worth becoming familiar with (though it also means that we're unable to use JavaScript's dot notation because it clashes with some other languages).

[6] http://mustache.github.com/

However, given that we're already using jQuery, we're in luck. In 2008, John Resig (the creator of jQuery) released a novel and extremely small templating engine made up of around 20 lines of JavaScript. It has since been expanded into a full jQuery library that is considered the officially sanctioned jQuery templating engine: the code is available from the jQuery repository on GitHub[7] and the documentation is located on the jQuery plugin site.[8]

To use the library, download the archive from GitHub and extract the minimized file. It will be called **jquery.tmpl.min.js**, and is only about 8KB in size. Copy this into your project to load it up:

```
<script src="jquery.tmpl.min.js" type="text/javascript"></script>
```

Twitter Integration with Templating

Having brought out the big guns—the jQuery templating engine—we need something a bit more juicy to use it on. The client has been toying with the idea of integrating some Twitter data into the app. Specifically, they noticed that Twitter does a good job of celebrity stalking too—and they want to harness that stalking power in their app.

The plan is to use a public search on the query term "celeb spotting," and display any tweets that match. Generally, you can't make cross-domain Ajax requests (it's a security issue—in fact, the same security issue that prevents you from placing Ajax requests to file:// URLs)—but Twitter provides data in **JSONP** format. JSONP is a common trick to grab data across domains by loading it inside a `<script>` tag. For this to work, we need to append the string `"callback=?"` to the URL request.

We've had a look at the Twitter API documentation,[9] and found the appropriate URL to conduct a search: http://search.twitter.com/search.*format*?q=*query*. All we need to do is call `getJSON()` with that URL and our search string:

javascripts/ch4/17-twitter-templating.js *(excerpt)*

```
var twitQuery = "celeb+spotting",
    twitUrl = "http://search.twitter.com/search.json?q=";

$.getJSON(twitUrl + twitQuery + "&callback=?", function(data){
  // Show some tweets
});
```

Now we have some data from Twitter! The tweets are stored in the `results` property of the `data` object as an array, so we can use them directly in our template system.

[7] https://github.com/jquery/jquery-tmpl
[8] http://api.jquery.com/category/plugins/templates/
[9] http://dev.twitter.com/doc/

Templates used in templating engines are defined in the same manner as the simple template we put together earlier: as elements inside our HTML file. The jQuery templating engine uses a clever trick to prevent the template from becoming a regular part of the DOM: it hides inside a `<script>` tag with `type="text/x-jquery-tmpl"`. Browsers don't know how to handle this type of "script," so they don't try to execute it, but we can still extract the content and use it as a template:

ch4/17-twitter-templating.html *(excerpt)*

```
<script id="tmpl-tweet" type="text/x-jquery-tmpl">
  <li>
    <a class="avatar" href="#"><img src="${profile_image_url}"
➥alt="${from_user}"></a>
    <a href="http://twitter.com/${from_user}"><h2>${from_user}</h2></a>
    <span class="details">
      ${text}
    </span>
  </li>
</script>
```

Template Types

Even if you're not using the jQuery templating engine, you can use the `<script>` trick with your own templates. You should use a different option to `x-jquery-tmpl` though; you can make up your own `type` if you want, but you'll often see the `text/html` used.

You'll notice that inside the HTML there are some odd-looking bits. These are are our tokens that will be replaced with data. The strings inside the curly braces aren't made up: they are the names of the values that are returned to us from Twitter. To find out exactly what data is returned by a given API call, you can either read the documentation, or simply log the data object to the console using `console.log(data);`.

All that's left to do now is to call the templating function. Just select the template item by `id` (we've called ours `tmpl-tweet`), and call the `tmpl()` method, passing the `data.results` array we retrieved via Ajax. The engine will see that our array has multiple items; for each item it will generate a clone of the template and replace all the tokens with the corresponding values from the data array. The result is a jQuery object containing the constructed DOM node, which we can inject into the page as we'd normally do:

javascripts/ch4/17-twitter-templating.js *(excerpt)*

```
$.getJSON(twitUrl + twitQuery + "&callback=?", function(data){
  $("#tmpl-tweet")
    .tmpl(data.results)
```

```
    .appendTo("#spots-list");
    transition("#spots", "show");
});
```

No need to do any looping or manual value insertion. The combination of data-rich APIs and simple templating engines gives you a lot of power as a developer, easily enabling you to enrich your applications with information from a variety of sources. A sample result from our Twitter search is shown in Figure 4.5.

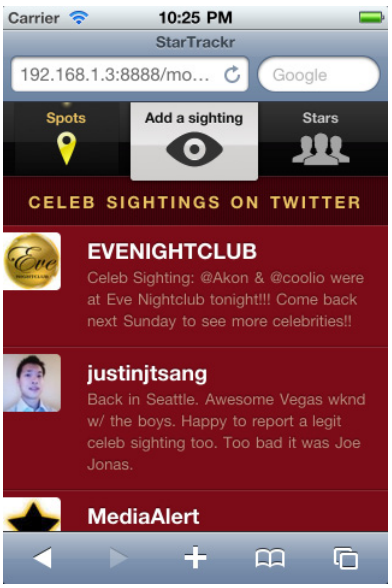

Figure 4.5. Pulling in celebrity spottings from Twitter

The jQuery template engine also includes tags for conditional evaluation like `{{if}}` and `{{else}}`, so you can show different parts of your template according to the situation. Check out the documentation for the full breakdown of available functionality.[10]

We Have an App!

We've taken our mobile website and transformed it into a decent mobile app that our users are going to love. Not too shabby, and we've only just picked at the surface. There's infinite room for creativity and experimentation, and a million ways we can tie everything together—we'll look at managing that soon. But be warned now: making mobile apps is addictive!

[10] http://api.jquery.com/jquery.tmpl/

Take the Next Step with Us

Now that you've read a few chapters, I'm sure you agree that it's exciting stuff. Yet, there's still more ...

This book will help teach you to build sites and apps for the latest generation of mobile devices, including smartphones and tablets—and stay on top of your game.

Why not take the next step, and grab the full version now!

Why order the book?

- **Full-color layout with instructive images**
- **Loads of examples with a fun, hands-on approach**
- **Available in print and for all electronic devices**
- **Free shipping when you order any other book**
- **100% money-back guarantee**

ORDER DIGITAL COPY

ORDER PRINTED COPY

100% Satisfaction Guarantee

We want you to feel as confident as we do that this book will deliver the goods, so you have a full 30 days to play with it. If in that time you feel the book falls short, simply send it back and we'll give you a prompt refund of the full purchase price, minus shipping and handling.