Microsoft

Architecting Mobile Solutions for the Enterprise

Dino Esposito

Architecting Mobile Solutions for the Enterprise

Expert guidance for planning and executing a complete mobile web strategy

Rethink your approach to the mobile web and native apps—and build tailor-made solutions to reach customers and clients on a variety of devices. Led by web development luminary Dino Esposito, you'll learn how to create an effective mobile strategy that meets the unique B2C or B2B needs of your enterprise. You'll also gain architectural and implementation guidance for building mobile-specific websites, native and cross-platform applications, and more.

Discover how to:

- Reach more users with a combination of mobile websites and platform-specific apps
- Architect a mobile-optimized website accessible from many different devices
- Use HTML5 and jQuery Mobile to build sites that look and behave like native apps
- Get started with the basics for building native apps for Windows® Phone, iPhone, and Android
- Implement design patterns specific to mobile application development
- Develop cross-platform app features, such as localization and offline behavior
- Write one application codebase for many platforms using the PhoneGap framework

Get code samples on the web
Ready to download at
http://go.microsoft.com/FWLink/?Linkid=247992

*For **system requirements**, see the Introduction.*

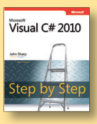
About the Author

Dino Esposito, CTO of a company that provides software and mobile services to professional sports, is an expert trainer and web architect. He's the author of several popular books for Microsoft Press® such as *Programming ASP.NET 4* and *Programming ASP.NET MVC 3*.

DEVELOPER ROADMAP

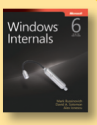
Start Here!
- Beginner-level instruction
- Easy to follow explanations and examples
- Exercises to build your first projects

Step by Step
- For experienced developers learning a new topic
- Focus on fundamental techniques and tools
- Hands-on tutorial with practice files plus eBook



Developer Reference
- Professional developers; intermediate to advanced
- Expertly covers essential topics and techniques
- Features extensive, adaptable code examples

Focused Topics
- For programmers who develop complex or advanced solutions
- Specialized topics; narrow focus; deep coverage
- Features extensive, adaptable code examples



ISBN: 978-0-7356-6302-2

microsoft.com/mspress

U.S.A. $39.99
Canada $41.99
[*Recommended*]

Programming/Mobile

Microsoft®

Microsoft

Architecting Mobile Solutions for the Enterprise

Dino Esposito

ISBN: 978-0-7356-6302-2

1 2 3 4 5 6 7 8 9 LSI 7 6 5 4 3 2

Printed and bound in the United States of America.

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at mspinput@microsoft.com. Please tell us what you think of this book at *http://www.microsoft.com/learning/booksurvey*.

Acquisitions and Developmental Editor: Russell Jones

Production Editor: Kristen Borg

Production Services: S4Carlisle Publishing Services

Technical Reviewer: Marco Bellinaso

Copyeditor: Sue McClung

Indexer: Margaret Troutman

Cover Design: Twist Creative • Seattle

Cover Composition: Karen Montgomery

Illustrator: S4Carlisle Publishing Services

To Silvia, because you're stronger than you think.
To Michela, because you're just the daughter I always dreamt of.
To Francesco, because you're a terrific, quick learner.

—Dino

Contents at a Glance

Contents

Introduction

As far back as 1999, some smart guys predicted that mobile would become the primary focus of development in only a few years. Although it has taken a bit more time than expected, the era of mobile software has arrived at last. Why did it take so long? The answer is surprisingly simple: mobile software needed a critical mass of users to develop before it could take off. The process of accumulating mobile users probably started with the release of the first iPhone back in 2007, but today, it has reached a large enough mass to trigger all sorts of chain reactions.

Back in 1990 (yes, you read that right), Bill Gates gave a keynote talk at Comdex titled "Information at Your Fingertips." Let's be honest—for 20 years, we pretended we really had information (that we needed) at our fingertips, but at most, we had that information only at hand—which makes a huge difference. Now is the time, though, that we can cover the short distance from hand to fingertips. With mobile devices everywhere, and especially with a revolutionary version of Windows on the horizon, I believe we're truly entering a new era of development—a paradigm shift.

Paradigm shifts just happen—and mobile represents a big one. Mobile enables new business scenarios and new ways of doing the same business. Mobile affects nearly everybody—users, professionals, and clearly developers. Writing mobile applications is a challenge that the vast majority of developers will face in the near future. Overall, mobile applications are simpler than desktop or web applications—but that's true only if you count just the number of functions. The hardest part of mobile development is to identify the right set of use-cases and the right user experience and interaction model. It turns out that the typical mobile application user is much less forgiving than the average user of web or desktop applications. As developers, we forced users to play by the rules of software for decades. In contrast, mobile developers will be forced to play by the rules of user experience and conform to user expectations. This is how software always should have been; but it's definitely not how software has been built for at least the past 20 years. Moreover, before too many more years pass, mobile may well be the only software that we will be called upon to write.

The term mobile refers to a variety of platforms, each with its own set of capabilities and features, and each of which requires significantly different skills: different operating systems, different programming languages, different application programming interfaces (APIs), and even different computers. A mobile application is more sophisticated and more complex than web applications with regard to resource management, data entry, sensors, data storage, and life cycle. Furthermore, each operating system has its own set of development guidelines and a proprietary deployment model.

This book is intended as a quick-but-juicy guide to issues that you may face while developing a mobile project for one or multiple platforms. The book starts by analyzing the various types of mobile solutions, which include websites, websites optimized for mobile devices, and native mobile applications, and then identifies a few design patterns common to all mobile applications and technologies available on the various platforms. Predictive fetch, back-and-save, and guess-don't-ask are just a few of the patterns being discussed and implemented. The book puts considerable emphasis on mobile sites and frameworks, and on techniques to detect browser capabilities accurately. For example, the book offers a chapter on Wireless Universal Resource FiLe (WURFL)—the framework being used by Facebook for mobile device detection—and compares that to the detection capabilities in plain ASP.NET.

Furthermore, the book offers an overview of mobile development for the three major platforms—iOS, Android, and Windows Phone. In particular, this book builds the same application for all three platforms, discussing tools, frameworks, practices, and illustrating architectural and structural differences along the way. Finally, the book covers PhoneGap and HTML5-based development for mobile devices.

After reading this book, you probably won't be a super-expert in any of those platforms, but you'll know enough to start producing code on any of the most popular devices. You'll also know enough to advise your customers and help them define effective mobile strategies for their business.

Who Should Read This Book

As companies start going mobile, they need a strategy long before they need a mobile site or an iPhone app. But when companies have developed the strategy and start looking into implementing it, they face the rough issue of not having or finding architects and developers that know the mobile world from a variety of angles. Today, they can easily find great iPhone or Android developers, but they can hardly find a consultant that can suggest, based on strong evidence, whether a mobile site is preferable for them.

This book is aimed at providing an architect summary of what you need to know to design and implement mobile solutions. Today, a mobile solution often means arranging the same application for several different platforms (iPhone, Android, Blackberry, and Windows Phone), and doing that using a very specific set of design patterns with little in common with desktop or web apps. Last but not least, the effort must be done in the context of the customer's needs, expectations, and existing business.

Not a Mobile Developer? Not a Developer!

For a company with a consolidated business, mobile is a way to expand its horizon. The new expansion stage of mobile is reaching out to companies and enterprises and prospecting new ways of doing business. This is a paradigm shift with a deep impact that will give rise to new professional jobs, much as the web itself did more than a decade ago.

That's why I maintain that in only a couple of years, every developer will be either a mobile developer or no developer at all. Being a mobile developer surely includes knowing iOS, Windows Phone, HTML5, and Android, and perhaps BlackBerry, possibly Bada, and even developing for smart TVs—and, of course, for the mobile web. More than anything else, though, developers must acquire a "mobile mindset." You can always figure out fairly easily how to play a video on iOS, or how to make an Android device vibrate. But what isn't as easy to acquire is the intrinsic nature of mobile applications and the patterns behind them, and which aspects to focus on for optimization.

Mobile is different. Overall, it's simpler, but it's also much less forgiving than other types of applications.

Therefore, this book is for everybody who needs to acquire some mobile development insight. The book's contents won't become obsolete in just a few months because I made a serious attempt to reach and report from the heart of the mobile experience. This book discusses technology, but it is not based on any particular technology; therefore, it's an introductory text for any form of mobile development.

Who Should Not Read This Book

This book won't make you a top-notch iPhone or Android developer; it's intended to help everybody (including those of you who are already top-notch iPhone or Android developers) understand the entire mobile world. The goal is to get readers prepared for architecting effective mobile solutions after a mobile plan has been finalized and accepted. If you're looking for detailed, step-by-step examples of how to play an animation, make the phone vibrate, or making an Internet call on all possible platforms, you won't usually find them here. But I hope that you will find enough to help you get started with every aspect of mobile development.

Organization of This Book

This book is divided into three sections. Part I, "Going Mobile," is about the possible strategies to approach the mobile world. Part II, "Mobile Sites," covers the architecture and implementation of mobile sites and also touches on HTML5 and jQuery Mobile. Part III, "Mobile Applications," is about the three major mobile platforms of today—iOS, Android, and Windows Phone—and also covers PhoneGap as a way of unifying development in a single codebase.

Finding Your Best Starting Point in This Book

The different sections of Architecting Mobile Solutions for the Enterprise cover a wide range of technologies associated with mobile development. Depending on your needs and your existing understanding of mobile, you may wish to focus on specific areas of the book. Use the following table to determine how best to proceed through the book.

If you are	Follow these steps
New to mobile and spent your entire career doing other software-related work	Read the chapters as they are laid out in the book.
A web developer looking into how to build mobile sites	Focus primarily on Part II.
A chief technology officer (CTO) or chief architect	Focus on Part I first, and then move to Part II and/or Part III, depending on whether mobile sites or mobile apps are more likely to be relevant in your context. But read the entire book anyway.
Familiar with mobile app development in one (or more) platforms	You might want to start with the chapters that cover topics that you are familiar with. These chapters are essential guides, so it is likely that you won't learn anything new there. But if you find that you miss some of the points discussed, then you've got something already from the book. Next, I suggest you focus on Chapter 7, "Patterns of Mobile Application Development," and Chapter 11, "Developing with PhoneGap."

Note This table simply attempts to provide some guidance on how to learn best from this book. In any case, I heartily recommend that you read all the chapters thoroughly.

Conventions and Features in This Book

This book presents information using conventions designed to make the information readable and easy to follow.

- Boxed elements with labels such as "Note" provide additional information or alternative methods for completing a step successfully.
- Text that you type (apart from code blocks) appears in bold.
- A plus sign (+) between two key names means that you must press those keys at the same time. For example, "Press Alt+Tab" means that you hold down the Alt key while you press the Tab key.

System Requirements

You will need the following hardware and software to set yourself up for development on the various mobile platforms and compile the sample code that accompanies this book:

- For iOS, you need a Mac computer with Xcode and the latest iOS software development kit (SDK). If you plan to use MonoTouch, then you also need to get at least a trial version of the product from *http://www.xamarin.com*. Note that to deploy applications on a iOS device, you also need to be a registered Apple developer enrolled in one of the Apple pay programs.
- For Android, you can use a Windows PC, preferably equipped with Windows 7. Note, however, that you can do Android development from a Mac or Linux PC as well. You can use Eclipse or the IntelliJ IDEA as your integrated development environment (IDE). You will need the Java SDK and the Android SDK installed. You don't need to be a registered developer to compile and deploy Android applications on a device.
- For Windows Phone, you need Microsoft Visual Studio Express for Windows Phone, as well as a Windows PC.

Code Samples

This book comes with a few examples organized as follows:

- Two ASP.NET websites configured to use WURFL

- The Guess application for iOS
- The Guess application for Android
- The Guess application for Windows Phone
- The HTML5 Guess application for PhoneGap

The sample code contains files that you can incorporate in your own projects using the tools that you prefer.

Many of the chapters in this book include examples that let you try out new material discussed in the main text. You can download all the sample projects from the following page:

http://www.microsoftpressstore.com/title/9780735663022

Follow the instructions to download the Amse.zip file.

Note In addition to the code samples, your system should have Visual Studio 2010 and Microsoft SQL Server 2008 installed. The instructions that follow use SQL Server Management Studio 2008 to set up the sample database used with the practice examples. If available, install the latest service packs for each product.

Installing the Code Samples

Follow these steps to install the code samples on your computer so that you can use them with the exercises in this book:

1. Unzip the Amse.zip file that you downloaded from the book's website (name a specific directory, along with directions to create it, if necessary).
2. If prompted, review the displayed End User License Agreement (EULA). If you accept the terms, select the Accept option, and then click Next.

Note If the license agreement doesn't appear, you can access it from the same webpage from which you downloaded the Amse.zip file.

Acknowledgments

It took me several months of deep dive to make sense of the many facets of mobile: the customer's angle, the developer's perspective, the architect's vision, and the myriads of devices, operating systems, SDKs, and products. Many friends helped me out along the way.

First and foremost, I want to thank Marco Bellinaso of Mopapp, who first introduced me to the world of mobile apps and then served as an invaluable technical editor for this book. Marco also tried to make me a fan of Objective-C, but I'm afraid his efforts failed in that regard.

Devon Musgrave of Microsoft Press and Russell Jones of O'Reilly believed in this book and made it happen, along with Kristen Borg and the other members of the editing team.

I was surprised to see how many friends asked to review chapters and enthusiastically shared their feedback. I could see an underlying passion and pleasure in their work and I'm not sure my monumental THANK YOU here is enough. In particular, I wish to thank Luca Passani of ScientiaMobile. I met Luca at a web conference in London in 1999, where he tried to sell me mobile as a hot business even back then. It took a bit more time, but his vision was definitely right. I really enjoyed the feedback about mobile site development and HTML5 that I got from Jon Arne Saeteras of MobileTech and Daniele Bochicchio of 5DLabs. IT and Microsoft Regional Director for Italy. The chapters on mobile apps and PhoneGap benefited from the feedback of many people, including Davide Zordan, Ugo Lattanzi, Leon Zandman, Catalin Georghiu, and Davide Senatore. All these people shared their real-world experience with me concerning Windows Phone and PhoneGap.

Near-final thanks go to my team at Crionet and E-tennis.net. As I write these notes, we are finalizing the mobile apps for the worldwide audience of tennis fans following the Rome ATP Masters 1000 tournament. It's the first tournament to offer a comprehensive mobile, web, and social experience and the first one to offer mobile apps on a full range of platforms, including not just iOS and Android, but also Windows Phone and BlackBerry. Working with you guys is a privilege.

What else? Well, just a final note. Take note of this name: Francesco Esposito. I'm sure you'll hear this name in the future. He's 14 and he's already an all-round mobile developer. My use of the word developer is no accident, because that's what he is, irrespective of schooling and age. In his way of coding, learning, thinking, and speaking, I see crystal-clear talent. Being his dad, well, I feel proud.

Errata & Book Support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site:

http://www.microsoftpressstore.com/title/ 9780735663022

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at mspinput@microsoft.com.

Please note that product support for Microsoft software is not offered through the addresses above.

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in Touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*.

PART I

Going Mobile

CHAPTER 1

Pillars of a Mobile Strategy

In preparing for battle, I have always found that plans are useless, but planning is indispensable.

—Dwight D. Eisenhower

In this chapter:

- What Does "Going Mobile" Mean?
- Outlining a B2C Strategy
- Outlining a B2B Strategy
- Summary

The modern era of mobile technology began with the release of the first Apple iPhone in the summer of 2007.

The mobile conquest of the world has been a "soon-to-be" matter for quite some time in the past decade. I still remember the first-ever mobile-related conference being held in Amsterdam in the summer of 2000—the Wrox Wireless Developer Conference. I was a speaker there, and the implicit message for attendees was "Mobile development is here—hurry up."

There was no hurry, actually.

Only a couple of years later, Microsoft released ASP.NET with its own set of mobile controls for optimized mobile websites. Later, mobile frameworks such as Microsoft .NET Compact Framework and Java Micro Edition (J2ME) appeared; meanwhile, richer native operating systems such as Symbian also appeared. However, the mobile conquest of the world never happened—and perhaps hadn't even begun—which begs the question: Why not?

The main reason is that the technology never reached a critical mass of users, and without that, developers and software houses had no good reason to address the mobile space. But when the Apple iPhone appeared, everything changed. Although the iPhone was not an entirely new idea, it was an extremely well-done implementation. And, more importantly, a lot of people (on the order of millions) liked it. That immediately created a breeding ground for new applications and gave mobile technology a new form and immediacy.

The lesson to learn from this is that software is the *effect* (not the cause) of the mobile phenomenon. People buy devices long before they have much compatible software to run on them. Therefore, a compelling device, bought by a critical mass of users, creates a compelling market for specific software over time.

Today, there are a few popular mobile operating systems and a growing number of users willing to pay to get nice applications to run on them. The popularity and convenience of mobile devices drives companies to create their own mobile applications that can reach their customers while they're traveling. Mobile sites are still an excellent way to do that, but whether companies build mobile sites or mobile applications targeted to a particular platform (today, that would include iPhone, Android, BlackBerry, and Windows Phone), companies need to be part of the mobile revolution in much the same way they became part of the web revolution a decade ago.

What Does "Going Mobile" Mean?

This book is aimed at architects and developers who are willing (or need) to implement mobile solutions for customers. A solution, however, is not necessarily and not simply a mobile application. Today, and even more in the near future, a *mobile solution* will be created as a combination of a classic website for desktop browsers, a website specifically designed for classes of mobile devices (known as an "m-site") and one or more applications for specific mobile operating systems.

The definition of a mobile solution is not carved in stone, for two excellent reasons. First, the mobile industry never sleeps; it churns out requirements and opportunities at an impressive pace, so any current definition of a mobile solution may change to incorporate new aspects in a matter of just one or two years. Second, a mobile solution applies to a particular business scenario. The business scenario ultimately determines the details of the solution and technologies, patterns, and platforms that architects and developers will deal with. As an example, you may need to add some Facebook applets or multiplatform desktop applications if the business has social networking implications. Similarly, you might restrict the range of mobile platforms to just one if you're building a vertical enterprise-class solution for a single customer.

As I see things, going mobile is a far more serious task than simply writing an iPhone application. Companies investing in mobile need a strategy long before they need a mobile site or a set of mobile applications. This means companies must establish goals as well as review processes for achieving those goals—simply put, they must have a strategy.

To paraphrase the quote from Dwight Eisenhower at the beginning of this chapter, in mobile development, plans are useless, but planning is indispensable.

Toward a Mobile Strategy

So the first step for a company "going mobile" is to define a strategic plan. The strategic plan is more conceptual than it is an operational plan with comprehensive implementation details. The strategic plan is visionary; it identifies the future direction of the business. Outlining a mobile strategy essentially consists of reviewing the current business processes with regard to a few mobile axioms.

Three Mobile Axioms

Gone are the days in which a website optimized for a bunch of desktop browsers was the only way for a company to deliver an application. Today, there's a growing demand for applications that users can reach from a variety of platforms and browsers. In the past, software architects once reached for the Holy Grail of multiplatform development—and we failed to grasp it. Now, as users increasingly demand multiplatform applications, failure is simply not an option.

Mobile axioms are statements about mobile applications that are self-evident and assumed to be true. You should have these concepts clear in your mind before you start planning your strategy:

- Provide your services through multiple channels.
- Look for new opportunities and new ways to provide your services.
- Aim at making your customers' lives easier.

Like the web a decade ago, mobile is about new ways of doing both a selection of old tasks and entirely new actions. Mobile is highly attractive to users because they can get the services they need in a variety of ways and using a variety of devices. As a company, "going mobile" means being committed to making your customers' lives easier through ad hoc and personal services.

The fundamental point, however, is that this challenge is not limited to just a few segments of the industry; it's a global challenge.

Multiple Channels

As you can guess, going mobile likely involves significant investments on your side to restructure existing processes, implement new ones, and fix—or at least extend—a portion of your back end software.

Delivering services to a variety of channels is challenging. Mobile channels (tablets, devices, or mobile sites) are more personal and typically involve smaller amounts of information. Your existing back end must be able to serve these new requests effectively while preserving both scalability and performance, and while still ensuring at least the same level of security.

A good example of an application delivered through multiple channels is Facebook; other examples are airline booking and home banking services.

New Ways to Provide Services

Mobile is both about bringing existing services to people's fingertips and about creating brand-new services. A mobile device is a personal device, so everything that shows up there is potentially "at your fingertips." The real estate of a mobile device is considerably smaller than a laptop, but most applications and websites are padded with extra information (including menus and layout) that is not necessarily required. The advantage of a mobile solution in this context is that it can provide exactly what's needed whenever the user needs it—instantly.

A mobile user is typically traveling around. Your application may query the user's current location and use that information to offer new, unique, and tailor-made services. Location-aware services are really at the heart of the extra power of mobile applications. This is not so much because a desktop site is unable to detect the user's location, but because a site can use the location details in much more compelling and useful ways when the user is out of the office. This is definitely an area to explore if your business is in any way related to location.

As an example, an application that provides information about transportation can use your location data to restrict search or sort results automatically, winnowing out nonessential data for other locations. The same concept applies to mass retail applications, which might notify users of special offers when they are close to a shop, or provide them with free coupons in a nearby shop that they can reach within a few minutes.

Simplify Customers' Lives

I see more and more companies from a variety of industry segments strongly committed to making their customers' lives easier and better. I believe that is a key challenge for attracting new customers and keeping existing ones. On the other hand, by not going mobile, you risk alienating customers from your brand.

As mentioned earlier, mobile applications are more personal than desktop applications. They're often relatively simpler in terms of logic and complexity, and they often consume smaller amounts of information. That's precisely what makes a customer's life simpler—the application is more focused; ideally, it can handle more related information aggregated from multiple remote sources. Basically, an effective mobile application should be able to give users what they need at any particular moment.

Architecting the system around these new needs is the effort that companies should invest in. It's not simply a matter of software architecture, though. Architects may be able to tell the best way of realizing an idea, but they can hardly identify what makes your users happier. In general, an appropriate analysis and prioritization of use-cases selects the range of features that—once implemented—put more information at the user's fingertips and make life easier.

Mobility and the Industry

According to a Gartner report presented in the spring of 2010, mobility occupies a relevant position in the list of top priorities for chief information officers (CIOs) of various industry sectors through 2013. According to this report, transportation and retail are the industry sectors that are paying the most attention to mobility.

In these sectors, there's a strong sentiment that it is an "either now or never" matter; there's less and less space left for companies that hesitate or just skip going mobile. The mobile space is open for business (for now) and companies need to establish their presence as soon as possible. If they don't, others will fill the gap and become your toughest competitors.

Also, according to Gartner, beyond transportation and retail, other sectors interested in mobility are healthcare, utilities, education, and—guess what—software publishers. Media and financial services are also there, lower on the list.

The trend that Gartner excerpts from CIOs' priorities may be different from country to country; however, past history shows that a general trend is always a trend that applies worldwide (though at a different pace in various locations).

I can contribute my direct experience in Italy, where most leading mass retail companies are only now experiencing what many experts call the first stage of mobility—merely establishing a static presence. Typically, this process is initiated via nearly functionless mobile sites that go hand in hand with existing primary desktop sites. The next step usually involves adding a bit of context through proactive alerts, and advertising based on location, identity, or perhaps barcode recognition. Finally, the third level of mobility awareness concentrates on providing all-round services at users' fingertips.

Defining a Mobile Strategy

Each business has its own mission, expressed as purposes and activities. A mobile strategy revisits and extends these purposes and activities in light of new devices and a new lifestyle. The mobile axioms should just inspire a realistic vision for the mobile business.

If this scares you, don't worry: it's nothing new—in fact, you've been there already, a decade ago. Although different in features and results, the mobile revolution follows the same pattern that the web revolution did. Early adopters content themselves with just being there and show customers they're online. Then, executives start developing a new vision of the business and architects actually build it. It's not a waterfall-like process; actually, it has a lot of inherent agility and looks like an intertwined process. In the end, every company ends up with what the management envisioned in their future—good and bad.

What Do You Want to Achieve?

Personally, I think that for most companies, embarking on mobile projects is not a choice related to gaining an immediate profit. Of course, that mostly depends on the type and size of company. If your business is selling ringtones, then naturally you expect profits from your mobile software right away. However, if your business is selling news, you might want to use mobile channels to make your readers' lives easier, so long as you can add such services at a reasonable cost to you. With the all-free model becoming less affordable every day, going mobile and attracting readers with mobile device capabilities is an immediate expense that hopefully will help achieve better results in the medium term or in the long run.

With a strategy defined in terms of expectations and requirements (covering growth, profitability, and markets), you can look at your overall mobile technology strategy. All in all, there are two (not mutually exclusive) possible expectations: reaching the largest possible audience and improving the experiences of existing customers by building a rich, jaw-dropping application. Implementing each scenario may require a different set of concrete technologies, languages, and platforms. And each scenario may have different costs.

Reach Out to Users

You reach mobile users by making your application available on the devices they use. This apparently obvious, no-brainer statement hides all the complexity (and costs) of mobile development. Take a closer look at this statement, though, and you'll find two huge questions whose actual implementation determines the actual level of complexity (and costs) of reaching out to users:

- Which devices are your customers using?
- How do you make your application available on all of them?

Before you can answer those questions, you need to think about this: What's a mobile device, anyway?

According to one widely accepted definition, a mobile device is one that you might have with you at any time, can be used more or less instantly, is a personal item, and can be used to connect to a network. A laptop, for example, seems to match most of these requirements—except that you are hardly likely to take it with you when you go out for a walk or buy groceries—and laptops don't usually start instantly. Cell phones mostly fall into the category of mobile devices (many cell phones have at least some browsing capabilities). Finally, smartphones and tablets match all the definitional requirements.

Note Recently, I used the preceding words, *more or less,* to introduce mobile development challenges to a developer audience. One of the attendees winked and playfully replied: "So, you mean that my Windows Mobile phone is not a mobile device? It takes ages to boot up."

A mobile strategy also depends on the level of control you can exercise over the devices your users have. For example, if in your context, *user* means "employee," then the company can decide to support just one mobile platform and focus development on that. If you think that *user* means "consumer," however, then reaching out to a large audience usually means developing multiple similar applications for various devices. The same applies to scenarios where *user* means "employee," but the company is giving its employees the option to use the device of their choice.

Deciding how to approach the technology is a delicate and critical point of a mobile strategy that I'll address in more detail in the section "Outlining a B2C Strategy," later in this chapter.

Offer Rich Applications

If you know that a significant share of your users connect to your site using a particular mobile device, or if the content you're offering can best be consumed on specific popular devices, then your mobile strategy should include the development of an ad hoc application optimized for that device. You don't have to target each possible family of devices; instead, you can establish priorities and add new applications progressively.

Suppose that you own a radio station. You want to increase your audience so you can sell more ad slots. Most radio listeners are faithful, so despite the switch to mobile, they may well still be listening to their favorite radio station while out and about. They might be listening via radio-equipped MP3

players, original equipment manufacturer (OEM)–applications using the embedded radio system of a mobile phone, or Internet-based free radio programs. In all cases, users can listen, but they can't interact and increase your site traffic. But if you can develop a specific mobile application and let listeners interact with your back-end systems via the web, consume streamed live music, access podcasts, traffic reports, news, submit feedback, blog, and more, you can gain interactivity and increase user participation.

Should you address all the major mobile platforms at the same time? That mostly depends on both your budget and management's expectations. One common pattern is to build an iPhone application first, and then follow that up with an Android or iPad application. At a radio station, to continue with the example, a tablet device such as the iPad may add little extra value compared to an iPhone. So the second step in your strategy probably would be to develop an Android application, letting iPhone and iPad users share the same application.

I'll return to this point in a moment and address it more specifically in the next chapter, but keep in mind that mobile applications don't necessarily mean iPhone or Android applications. A mobile site can be as functionally rich, and it is usually more cost-effective.

B2C and B2B

The full spectrum of mobile applications falls into one of these two categories:

- Business-to-Consumer (B2C)
- Business-to-Business (B2B)

A third label is worth mentioning, though: Consumer-to-Consumer (C2C). Although not terribly relevant at the current stage of the industry, C2C provided the spark for the whole mobile revolution. The mobile revolution we're experiencing these days would probably have remained on hold for another 10 years without a lot of (initially) independent developers who enthusiastically embraced iPhone and Android programming and built clever applications (regardless of their usefulness). Some of these developers capitalized on the success and exposure of a single application to build a business and help the mobile revolution thrive.

Going B2C or B2B poses different challenges and drives different implementation choices. For example, in a B2C scenario, a key decision is about how to make the application available and get consumers to notice it—whether it's a free or paid application. In some cases, the question is a no-brainer (the app pretty much has to be free). In other cases, a more sophisticated model that offers a free (but perhaps feature or time-limited) version of the application is offered to entice users to purchase the full-feature paid version. In still others, consumers can select either an ad-supported version or an ad-free paid version.

In contrast, in a B2B scenario, you have a fixed number of users to reach. Here, your focus is on enabling users to return what you expect quickly, effectively, and securely. Security and middleware, in fact, are usually far more important in B2B scenarios.

Development and Costs

Developing mobile applications is neither cheap nor quick. Many companies find this surprising when they approach mobile projects. But mobile development is only *apparently* similar to web or Windows development; the two have different programming frameworks and often different (and uncommon) programming languages. Furthermore, mobile suffers from the lack of a consolidated set of patterns. Another reason that raises costs for mobile is the need to produce different user interfaces (often both layout and images) for different devices. This has never been a requirement for web or desktop applications. All these factors currently make mobile development significantly *more* expensive and time-consuming than web development, although time will help alleviate some of these issues.

It is commonly believed that outsourcing development is preferable to having in-house development, largely because in-house development means that you first need to invest in training. It's one thing to train a team of developers on ASP.NET and then have them build three sites in a row. But it's quite another to train a team on three different mobile platforms and then have them build the same application three times from scratch—once for each relevant platform you plan to address.

Outsourcing allows you to eliminate in-house training costs and speed up development. In return for this, however, you must pay more for the outside expertise. It's worth exploring some of the reasons that make mobile development more expensive than many executives think at first.

Targeting Multiple Platforms

The mobile ecosystem is populated by several different platforms, each of which has its own more-or-less unique set of features and capabilities. The most popular platforms today are iPhone, iPad, Android, Windows Phone, and BlackBerry. The list of platforms, however, doesn't end here. Other platforms that you are likely to encounter or need to consider are Symbian, Windows Mobile, Meego, Bada, QT, and webOS. And when you begin to look at using tablet devices, the range of platforms that you may need to take into account grows even more, because there are tablet-specific variations of the aforementioned platforms, including Android Honeycomb, BlackBerry PlayBook, and the upcoming Windows 8.

Each platform has its own operating system, its own programming application programming interface (API), and its own set of programming guidelines. Often, each mobile platform requires applications be written in a specific programming language, such as Java, Objective-C, C#, or C++.

So does this mean that you must port or develop your application from scratch for each of these platforms?

Frankly, very few applications (e.g., content providers) need to address all these platforms. More typically, applications target a subset of no more than three or four of them. If it is crucial for your business to reach the largest possible audience, even those running on low-end devices, then you might want to look at HTML—specifically HTML5—to build a website optimized for mobile devices (i.e., an *m-site*). As you'll see in more detail in the next chapter, m-sites are often the first option that you should consider when targeting multiple platforms is a true business necessity. M-sites, however, are not free of device issues either. In the end, building a mobile site can be considerably more complicated than building a website.

Addressing the Device Fragmentation Issue

If you felt frustrated by desktop browser fragmentation—too many different browsers to optimize webpages for—you have never explored the mobile jungle. Each device—and by device I don't simply mean smartphones—has its own browser, and each browser has its own user agent string, which changes for each version and operating system update. And, of course, the actual set of capabilities can change for each device as well. The screen size is probably the most important capability to take into account because of real estate and pixel density.

The dimension of the device fragmentation problem is far larger with mobile browsers than with desktop browsers. When it comes to mobile site development, you have thousands of different device models to take into account, not just a few dozen smartphones, often with a pre-fixed set of capabilities. How can you approach such a task?

Writing a set of pages (if not the entire site) on a per-device basis is simply not feasible. The one-size-fits-all approach is viable, but it comes at the cost of leaving a lot of older devices behind and giving up on advanced features that smartphones have. This is typically not good enough for companies whose success depends on online content, such as social networks, or media and news companies. The alternative is *multiserving,* which basically consists of three points:

- Group devices in classes based on their capabilities
- Build a version of the site for each class of devices that you intend to support
- Define a strategy to serve the right site for each connecting device

That's easy to say, but how can you determine the capabilities of a given device? How can you know the size of the screen, the operating system, the quality of video codecs, whether the device supports graphic processors or certain HTML features (e.g., file upload and CSS gradients), the availability and accuracy of location services, and even much more specific capabilities, such as image inlining (the ability to display images from page-embedded Base64-encoded strings)?

For some of these capabilities, such as screen size, you can ask the browser itself. In fact, forums are full of questions about how to determine effectively the "real" size of a screen on a particular device and model. For other capabilities, such as image inlining, there's just no way to make such a query. You just must *know* it.

About 10 years ago, Luca Passani had the vision of starting a community-driven project aimed at collecting reliable information about the effective behavior of mobile devices. He created the WURFL project, short for "Wireless Universal Resource File." Today, WURFL is a centralized database that stores detailed information (more than 500 different capabilities) about more than 15,000 mobile devices and mobile browsers. Today, WURFL is managed by ScientiaMobile (*http://www.scientiamobile.com*) and made available through both commercial and open-source licenses.

Multiserving takes mobile development to a new level of complexity, but this is where WURFL shows its value: WURFL makes multiserving manageable. Multiserving is inherently expensive, but using WURFL can make it considerably less expensive.

I'll return to the topic of mobile site development in Chapter 4, "Building Mobile Websites," and cover WURFL features in detail in Chapter 6, "Developing Responsive Mobile Sites."

Note WURFL is the device detection engine that powers a number of very large and popular mobile sites: Facebook, Google, AdMob, and a long list of mobile network operators and virtual network operators.

Looking for Best Practices

If you are building a desktop website, you can rely on a number of tutorials, widgets, articles, books, and posts that give you guidance. The same isn't true for mobile software.

The importance and complexity of mobile site development is not yet perceived in its entirety. Too many developers (and, worse, architects) succumb to the siren call that m-sites are simply standard websites with different Cascading Style Sheets (CSS) and layout.

Turning to native mobile applications, all you can find are official API references, long and staid official guidelines in the form of white papers, and a ton of useful tips and tricks scattered in a variety of question/answer sites (such as StackOverflow). This is largely because mobile applications are relatively new and the entire space is fragmented; very few developers who program for iPhones know (or are interested in) Android or Windows Phone development. Furthermore, the stereotypical iPhone/Android developer considers mobile sites old-fashioned.

The bottom line is that when you are facing mobile development for business (for example, say your boss told you that you have to build an application in just a few weeks), you have no good place to look for common practices. Even when you can figure out most common practices, it's tough to know whether those common practices are also best practices.

The Marketplace Tax

Finally, development of mobile applications is subject to appstores. Apple made this model popular with i-tools (such as the iPhone, iPod Touch, and iPad); Microsoft took the same route with Windows Phone (and seems to be inclined to forge ahead with it in Windows 8); Google (for Android) and RIM for BlackBerry left their appstores optional for developers.

The role of appstores is crystal clear: they are there to protect users who buy or download applications from an appstore to their devices. The appstore owner guarantees the quality of published applications. For developers, getting approval from the appstore owner requires more effort to ensure the quality of the final product—which is not a bad thing for consumers. For companies, the appstore model means that there's an extra distribution cost, which I like to call the "marketplace tax." Companies have to pay to gain the right to distribute even free applications, and for paid applications, they typically have to provide about 30 percent of the app's revenue to the appstores.

Outlining a B2C Strategy

A B2C strategy is built around two pillars: reaching out to users and making them happy. Both pillars are quite generic and can be implemented in various scenarios with slight variations.

You may need to reach the largest audience possible, including holders of low-end devices devoid of flat connectivity rates. Likewise, you may need to focus on holders (and potential holders) of smartphones. You may need to push a mobile application with certain characteristics to keep existing users and make them glad that they chose your brand. Alternatively, you may need a mobile application to attract and engage new users by offering new services or new ways of consuming existing services.

Needless to say, a B2C approach is particularly suitable for companies that already operate their core business in B2C mode. It comes as no surprise that, according to the Gartner report mentioned earlier, the industry sectors most interested in mobile are transportation, retail, healthcare, software publishers, financial services, media, and in general, content providers.

Focus on Your Audience

Any business that aims at being successful should focus on its potential audience and make projections about the composition of this audience in terms of age and other social and personal aspects. With this consolidated information in hand, you can make better plans. In this regard, a mobile strategy is merely a specific form of business strategy.

A mobile audience is made up of people who own a mobile device and are (or may be) interested in the services you provide. Figure 1-1 depicts these two sets of users and shows how mobile applications fit in with your existing customer base.

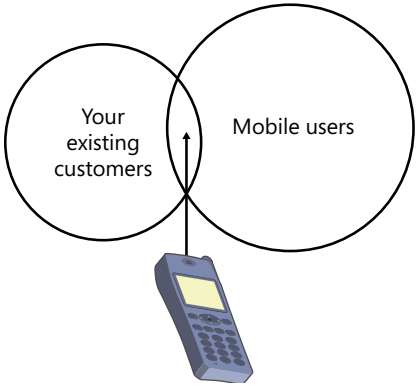


FIGURE 1-1 Mobile applications as the point of contact between existing customers and mobile users.

Note With regard to Figure 1-1, it should be noted that the overlap between "mobile users" and "existing customers" is moving and may change from month to month. When looking at the figure, don't take the size of overlap as truly representative of all businesses. The fact that the overlap is not null is perhaps the really important thing to remember.

Not all of your existing customers will become users of your new mobile infrastructure, but some generic mobile users will join the universe of your customers because of the mobile framework. This also should be read the other way around: If you don't go mobile, you may lose a share of your existing customers who are also mobile users.

A Quick Look at Global Numbers

It may sound obvious, but I'm going to say this anyway: the world is full of mobile devices. For the most part, these are low-end devices with a basic HTML browser, a quarter VGA (QVGA) screen (240 x 320 pixels), perhaps a camera, an MP3 player, and a few games and utilities.

According to the 2010 statistics of the International Telecommunication Union (ITU)—the agency of the United Nations (UN) responsible for information and communication technologies—there are 78 mobile devices per 100 inhabitants distributed all over the world, and a peak of 114 per 100 inhabitants in developed countries (see *http://www.itu.int/ITU-D/ict/statistics*).

Whichever way you look at it, the data shows that there are a few billion mobile devices of any type out there. How many of these are devices (and users) that you want to reach with your application? Probably as many as possible if you're Facebook or Google; a small fraction is enough otherwise.

The same ITU source reveals that there are about 30 Internet connections per 100 inhabitants all over the world, and 70 per 100 in developed countries. Although the two numbers are not directly related, this statistic gives a better approximation of the size of a potential mobile audience. However, according to eMarketer (*http://www.emarketer.com*), in 2011 the smartphone penetration in the world expressed as a percentage of all mobile devices is around 11 percent. That figure is expected to grow to about 50 percent over the next three years.

The data is more interesting when you look at these numbers for selected areas and countries. For example, the smartphone share grows to 37 percent in North America and 32 percent in Western Europe. It's around 10 percent in Asia and stays below 5 percent in Africa and Latin America. Amazingly, the country with the highest penetration is Italy, with 47 percent currently (expected to grow to 67 percent by 2014). And this in a country—my country—that still has wide areas of digital divide, and where one family out of three doesn't even have a home broadband connection.

The next section presents a few more numbers to help you understand the big picture of mobile connectivity.

A Deeper Look at Numbers

If you take global numbers literally, then by focusing on an iPhone application and disregarding mobile sites entirely, you cut off 90 percent of the potential worldwide audience—and even more than that if you consider that not all iPhone devices may be capable of running your application because of versioning issues. From this perspective, a mobile site seems to be a very reasonable choice.

Note That iPhone users are approximately 10 percent of the total smartphone-using population is an estimate that seems to find many direct and indirect confirmations from a variety of sources. Considering only the U.S. market, iPhone users represent about one-third of the smartphone segment, which is reported to range around 30 percent of the total audience for mobile devices. Statistics, however, depend on a number of factors and often represent little more than an opinion!

Regardless of your final choice, blindly looking at global numbers is not necessarily the correct approach.

Suppose that after running a few customer surveys and having analyzed your website logs, you know that 50 percent of your real customer base use iPhones and connect from Italy. Given those figures, should you really focus your effort only on a mobile site? Probably not. A desktop site that looks decent on most devices, that looks good on iPhones, and features a native iPhone application is the best combination. Note that the costs of implementing the iPhone application dwarf anything else.

On the other hand, if your business is selling ringtones or news, then you need to reach out to the widest possible audience, regardless of the devices they're using. A solution that reaches this objective with the lowest cost is your Holy Grail. Today, this means developing a solution based on HTML and JavaScript.

Facebook Was Not Built in One Day

In mobile, as well as in any business, time to market is critical. In laying out your strategy, consider applying an agile schema that lets you release applications piecemeal. Figure 1-2 presents the canonical Scrum process adapted to mobile projects.

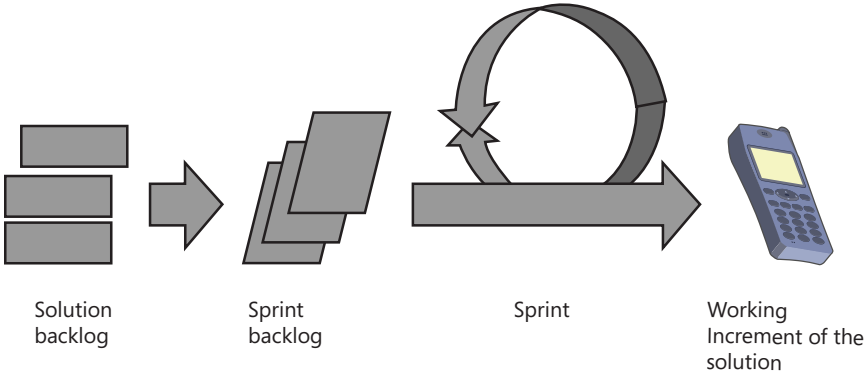


FIGURE 1-2 A Scrum-like model for mobile solutions.

The entire set of features and applications ("product backlog," according to the Scrum dictionary, and labeled "Solution backlog" in the figure) is partitioned into multiple sprints or iterations. At the end of each sprint, you release a working segment of the entire application (such as an iPhone application) and then are ready to reiterate the same process for another sprint (for example, an equivalent Android application).

More often than not, sprints for mobile solutions also include the following:

- Arranging a website that's usable by both mobile applications and sites. This means exposing the core functions of the website as easily callable, Representational State Transfer (REST)–based, HTTP endpoints. For example, if you're building the website using the ASP.NET Model-View-Controller (MVC), this may mean exposing an ad hoc controller that can serve requests based on the use-cases that you implement in mobile clients.
- Developing a set of pages (scripts, styles, graphics, and presentation logic) for a class of mobile devices. You may want to start with high-end devices and proceed downward to enable more and more lower-end devices to access some fraction of the full site functionality.
- Optimizing the behavior of these pages with more accurate device-detection capabilities.
- Developing native applications for most popular mobile platforms.
- At each level, you can propagate valuable user feedback through the entire stack of applications you've built thus far.

To paraphrase a popular saying, "Rome was not built in one day." I'd say that Facebook was not always the huge platform we know today, either—after all, it's been around only a few years. A mobile solution, therefore, will look increasingly like a small platform of integrated services; it requires hard work and overcoming many challenges to complete.

Delivery Models

A B2C application is (ideally) distributed worldwide. The costs of spreading the word about its availability (not advertising...) are entirely up to you. A website is immediately available from any place in the world, but again, the costs of spreading the word about it must be borne entirely by your organization.

In the mobile world, appstores rule over the publication and distribution of platform-specific applications. You publish your application to the appstore, giving it instant exposure to users of a particular platform. Each device ships with an applet so that users can access the platform's appstore—where your application gets published. Users can then access your application, read release notes, check requirements (such as that your application requires Internet access, phone calls, text services, local storage, and so on), see some screenshots, test-drive a trial version (if available and supported)—and what then?

What do you expect the return from investing in a mobile application to be? More generally, how do you expect to recover the costs of developing a mobile site and/or a few native applications? That's another part of overall strategy that management has devised.

Note Here, I'm talking about "spreading the word" and "publishing," which you get for free for the minimal costs of being a registered developer with the platform of choice. Advertising your application in and out of the appstore is another story entirely.

The Free/Paid Dilemma

Mobile applications are typically very cheap when they're not entirely free. The cost of the average iPhone application is around $2—even less for games. The average iPhone user is expected to download (and pay for, if that's required) about 80 applications in the course of a year.

Paid applications generate direct income subject to the marketplace tax (and, of course, government taxes). Free applications are generally built for marketing and branding purposes, or as an additional form of customer service.

After reading analysis and projections, expert opinions, and analytics, I formed the idea that mobile applications should be free; they need to generate revenue in some other way. However, if you're an individual or a small company and happen to have a stand-alone (not bound to a strategic business plan) mobile application, why give it away for free? If it's a well-done application that fills a hole in people's mobile lives, you can likely recover your investment, and perhaps even more.

A third option is advertising-supported applications, which are free for users but generate revenue for the author through dynamically inserted ads. Switching to a paid or ad-based model is an important step. If you first release the application for free, you get a lot more downloads, which are good for feedback. It also helps you understand how well received your application is and whether it really fills a hole.

If you look into the most popular appstores such as the Apple App Store, Android Market, Windows Marketplace, and BlackBerry App World, you will find that there are almost always more paid applications than free applications. For example, in the Android market, free applications outnumber paid applications by about a 60/40 ratio.

The free/paid dilemma is not really a dilemma with a binary, black-or-white answer. There are a few other models that mix free and paid content according to different recipes.

The Freemium Model

The Freemium model is based on the idea that you provide the full application free and then offer users the chance to buy a few extra services. From a realistic business perspective, however, the Freemium model means that the vast majority of your users will consume your application for free; only a minority will pay for any extra services.

So how can this model be worthwhile (financially speaking) for mobile applications? First and foremost, you need a lot of users, preferably on the order of millions, and at minimum on the order of tens of thousands. Maintaining all these users probably has a cost as well. For example, if you need to maintain a website to provide data to the mobile application, then you have a growing cost directly related to the number of users. Even if your application can run as a self-contained device application, you still may have some costs per user because you have to support users and reply to their emails.

An excellent example of a mobile application for which the Freemium model is perfect is Evernote (*http://www.evernote.com*). These mobile applications work entirely on the devices they target; all they need is storage space. According to *http://blog.evernote.com/2011/01/04/evernote-2010-a-year-in-stats*, Evernote has more than 6 million users. Of those, only 3 percent pay an extra subscription fee.

Another example is Searcheeze (*http://www.searcheeze.com*), a new startup that offers collaborative search. Users, both groups and individually, can run and publish a search on a given topic. This search—realized by humans, not search engines—may be left free or published to an internal marketplace. Becoming a Searcheeze user is free unless you want to buy extra services, such as installing a private engine on your company's servers.

The Premium-with-Free-Sample Model

The premium-with-free-sample model is fairly new in the media industry, but it's already the model toward which most content providers and newspapers are moving. Basically, it consists in making a significant portion of the content available for a small fee but leaving a fraction of the content free for everybody to access.

The New York Times pioneered this model. It currently gives you a number of free articles per month, after which you have to pay a fee to access more content. In contrast, the *Boston Globe* locked three-quarters of its digital content and offers free access only to the remaining part. Repubblica.it, Italy's largest news site and second-best-selling newspaper, also uses this latter model. In addition, Repubblica.it charges for access to its mobile site. In contrast, the desktop site is free, but you need a smartphone to read it effectively.

It's worth noting that the *Boston Globe* mobile solution is based on a HTML5-powered mobile site, which maximizes the audience without incurring the costs of developing ad hoc mobile applications and, importantly, without paying the typical 30 percent marketplace tax to an appstore owner. Appstores, in fact, may impose ad hoc policies for in-app payment. During the summer of 2011, Amazon quickly modified the Kindle iPhone and iPad applications to comply with new Apple policies for subscription-based applications. At nearly the same time, the *Financial Times* application—a best-selling program—was pulled from the store because it was patently in violation of the store rules. As a result, the *Financial Times* now encourages customers to use its new HTML5-based mobile site, which—guess what—has been optimized for iPhone browsers and looks nearly the same as a native application.

The Quid-Pro-Quo Model

As an Italian, I would have used another Latin phrase to express the same concept: *do-ut-des*. According to Wikipedia, the English usage of quid pro quo in fact matches the Italian usage of *do-ut-des* perfectly, meaning "I give so that I can receive."

This model is probably the one I feel most comfortable with. In my personal vision of the world, a mobile application exists as a complimentary feature, a favor that the publisher does for me. I reciprocate the favor by buying some of the publisher's other content or services.

The free applications are entirely free; there are no strings attached. To use them fully, however, you need to buy or consume some other services that the publisher relies on for income. Applications you use in an airport, during a tennis tournament, or at a conference are all examples that fall in this category. You get some services via the application in exchange for the simple fact that you're there (in airports or at conferences): you don't pay directly for these services (mostly information and news),

but you pay in some other way. For example, you probably paid to attend the conference or bought a ticket through that airport.

Here's another example of an application that I had the pleasure of knowing from an insider's perspective: I ported it from iPhone and Android to Windows Phone. The application is called Postino; you can find out more about it at *http://www.postinoapp.com*. Postino was originally built for the iPhone and then ported to a number of other platforms, one step at a time, as the result of a classic B2C strategy.

Postino lets you snap a picture as you travel and promptly creates a (virtual) postcard that you can send to a friend. The postcard contains a message, a signature that you draw on the screen with your finger, and an address. If the address is an email address, everything is free. If the address is a physical address, then you must buy a virtual stamp and upload the card to a server, which will print and send a real postcard.

An application built around a simple but good idea can be free, generate income in an indirect way, and still represent a success story for the developer (or the company), which may generate more business.

Outlining a B2B Strategy

I certainly don't have the expertise and experience to embark on a comprehensive discussion about the differences between B2C and B2B. As far as mobile strategies are concerned, there's only one important difference: B2B often gives you the chance to choose one specific platform and vendor and stick to that. From a software company perspective, B2B means that you're helping another business set up a mobile infrastructure that will be used to serve a limited and largely controlled audience, such as the network of agents that operate in a given region.

For the purpose of this book, the difference between B2C and B2B is the same as the difference between a public Internet site and an intranet site.

Serve Your (Limited) Audience

Let's review the main traits of a strategy aimed at serving the needs of just one business. The mobile interface is not open to the public; it's consumed by special customers, such as employees, agents, and consultants. Although you don't have to capture a large audience, you instead have a relatively small audience that you must serve in the best possible way. Forcing them to use one particular device or site is part of the deal.

B2B and the BlackBerry Case

What made BlackBerry so successful and BlackBerry devices so widely used? Sure, it offers email, tasks, and calendaring; it may even support web browsing and a camera. You can do some instant messaging and run a few utilities from an appstore that is one-tenth the size of Apple's. But compared to, say, an iPhone, a BlackBerry device looks like a child's toy.

So why was it so successful (at least before the iPhone arrived)?

The answer lies in the enterprise-class features that it offers. In particular, a BlackBerry device can connect to an in-house enterprise server—the BlackBerry Enterprise Server (BES)—and receive email updates, news, and task alerts in real time. How is that different from today's Microsoft Exchange Server connectivity in Windows Phone? It's not, really; both are basically the same—but BlackBerry was available somewhat earlier, and companies liked its features. As a BES administrator, you can apply policies and prevent a class of users from using the camera or instant messaging; you can force them to use only certain applications or to navigate only certain sites. Moreover, you can install your applications directly to your BlackBerry devices; you don't need to distribute them publicly to an appstore first.

In a nutshell, BlackBerry was a platform created to help members of an organization collaborate with ease and effectiveness.

Pick One Mobile Vendor

In a B2B scenario, a customer calls the software company and discusses requirements. The advisor has to figure out just one solution that provides the requested services in a mobile way. Most of the time, there are no constraints on existing devices and hardly any constraints to address on the platforms.

If you need a better mobile infrastructure to make employees collaborate, you probably have no reason to build an iPhone application. In addition to the costs of development, you also need to account for the costs of providing an iPhone to all your employees. I can think of a few companies who just did that—but I consider them the exception rather than the rule.

So in a B2B scenario, you should select just one vendor and platform and stick to that. From the customers' perspective, costs are clearly lower, and development time is traceable. Which vendor you settle on depends on a number of factors, including the existing base of devices, deployment needs, special security or middleware constraints, existing skills, and, of course, overall cost and personal preferences.

I'll return to this in a moment, but I think it's important to call attention to that point, because in a B2B scenario, the mobile vendor is not simply—and not necessarily—the vendor of a mobile operating system and API. In some cases, the candidate vendor doesn't even have its own mobile operating system. Instead, it offers its middleware with a bunch of platform-specific presentation layers for users to consume data and applications. According to Gartner's Magic Quadrant for 2010, Sybase is an excellent player in a B2B scenario—and Sybase doesn't have an operating system; instead, it provides a strong and powerful middleware for mobile clients.

Private Applications

When a company's goal is to build mobile solutions for its workforce, any applications that it develops should be private. A private mobile application is a mobile application that can be installed directly on one or more devices, with no intervening appstore. Consider, for example, an iPhone application written to serve the needs of a particular customer of yours—such as an application for sales agents.

That application is likely built to reflect the use-cases and business processes of that customer. It may have integrated some strong authentication policy. You don't want it to go to the marketplace,

and you don't want others to even look at it, let alone try it or buy it. You want it to work like Windows—you create an application, prepare an installer, run the installer on the machines you want, and that's it—you're done.

Private mobile applications are possible, but the process is not identical across the various platforms. In this regard, Android and BlackBerry are open: you can install just any application on just any device. For BlackBerry, this freedom of installing applications can be controlled and restricted by BES administrators. In Android, the only controller is the owner of the device.

Apple has a special enterprise program that, at the cost of $299/year, allows you to distribute applications freely within the members of your organization, whether through an intranet webpage, a network share, or email channels. Windows Mobile—the predecessor of Windows Phone—is as open as Android; Windows Phone still lacks an enterprise program. Currently, the recommended approach for simulating a private, company-wide marketplace is to make the application public and free and implement logic that unlocks the application only for users who have a specific Personal Identification Number (PIN).

Mobile Enterprise Application Platforms

In a B2B scenario, you typically choose a mobile vendor by analyzing its mobile enterprise application platform (MEAP). A MEAP indicates the entire stack of mobile technologies, products, and services that a mobile vendor (e.g., Sybase) offers.

MEAP vs. Stand-Alone Applications

When building a mobile solution, you could proceed by building a few stand-alone front-end applications that are based on an existing middleware or an ad-hoc back end and storage layer. But in doing so, you will likely end up using tools, services, and technologies from different vendors for the various phases of development.

MEAP is beneficial because by choosing a particular vendor, a company can often build a single back end and front end and deploy them to a variety of devices. The mobile device functions as a terminal that simply mirrors the content generated by the back end. A MEAP-based solution relies on proprietary middleware that you can customize and extend by writing applets using a few programming languages. The middleware serves data to the mobile client and controls both the user interface and local in-device logic.

With a MEAP in place, a company can expand its horizons with less effort—no need to invest in writing a new iPhone application; just deploy the same MEAP-specific application to the iPhone presentation layer. No changes are required to the underlying business logic, and the list of mobile front ends can be extended whenever the MEAP adds support for a new mobile platform.

In other words, a MEAP is an all-round business partner that specializes in mobile solutions. In this context, the classic iPhone or Android mobile application is just the tip of the iceberg—the real meat and potatoes are what lies under the surface.

Gartner's Rule of Three

To explain the importance of MEAPs and, at the same time, give companies an easy way to check their affinity with a MEAP solution, Gartner developed the *Rule of Three*.

According to Gartner, a company should consider a MEAP seriously when the implementation of its mobile strategy requires three or more mobile applications for three or more mobile operating systems to be integrated with three or more back ends. It goes without saying that building a mobile platform from scratch with these requirements is a huge effort that probably requires a monstrous budget. In this context, a MEAP can introduce significant savings and—more importantly—keep the company at the forefront of the technology, ready to release new products in a fraction of the time that non-MEAP-using competitors might require.

MEAP and Gartner's Magic Quadrant

How do you evaluate a MEAP? And more importantly, which vendors are actually MEAPs? Each year, Gartner applies its proprietary Magic Quadrant methodology to competing players in a given area—in this case, MEAP. The result is a diagram like the one shown in Figure 1-3.

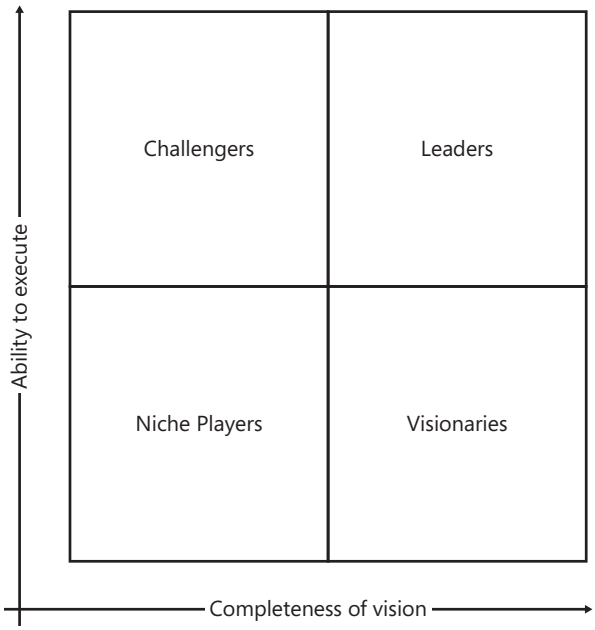


FIGURE 1-3 Gartner's Magic Quadrant.

The rank the research returns for each evaluated player determines the coordinates in the diagram and, subsequently, the quadrant into which that player falls. In the paper published in 2011, the Leaders quadrant contains companies such as Sybase, Antenna Software, Syclo, RhoMobile, and Pyxis Software.

It is worth noting that, according to Gartner, the MEAP market is steadily heading for a $2 billion volume in sales. So where are the other big companies commonly associated with mobile solutions? To be a MEAP player, a vendor must have a comprehensive set of products and services to develop and test applications and offer security, (cloud) storage, notification, reporting, and synchronization. Microsoft, Apple, and RIM appear in Gartner's Magic Quadrant but are not considered leaders in this segment.

Summary

No company can afford to ignore the mobile revolution taking place. Not all companies should proceed at the same pace or immediacy, but going mobile is a growing need and will soon be a necessity. The expression "going mobile" refers to the process of defining a strategic plan that sets business objectives that can be reached by restructuring internal processes, adopting innovative technologies, and developing ad hoc new applications to reach users who are traveling or to let one's workforce operate efficiently while away from the office.

This chapter outlined the main aspects of a mobile strategy both in a B2C and a B2B scenario. The next chapter takes a closer look at the two main ways of providing a mobile experience to users, whether customers or employees: mobile websites and platform-specific native applications.

CHAPTER 4

Building Mobile Websites

Intelligence is the ability to adapt to change.

—Stephen Hawking

In this chapter:

- From Web to Mobile
- Development Aspects of a Mobile Site
- The Device-Detector Site
- Summary

A mobile site, much like a native mobile application, is most likely to be consumed by users on the move: people who are standing up, in a hurry, busy, or waiting in line. Under these conditions, users are probably willing to connect to a site using a tiny device because they really believe that they derive some benefit from the site. The site, therefore, must be direct, concise, and accurate.

It is essential that a mobile site should load quickly and allow users to reach all main functionalities in just a few clicks or taps. The user interface should be extremely clear, but also clean and flawless to show the options available at any time, yet still making the act of choosing an option easy. After becoming familiar with the site, users often end up working with it semi-automatically, so even the position of a single button can have an impact on the quality of feedback that you receive. Note that the mobile Human-Computer Interaction (HCI) research field, although new, is very active, and a lot of studies exist about the dos and don'ts of interaction between mobile users and mobile devices and software. Luca Chittaro has a paper that effectively summarizes what mobile HCI means to developers and architects. You can read it here: *http://goo.gl/lSG3s*.

The previous chapter emphasized the importance of accurately selecting the use-cases to implement. The *number* of use-cases, however, should be kept small so that the site doesn't end up as a shrink-wrapped version of the full site. A pragmatic (and then not necessarily exact) rule is that a mobile site rarely needs more than 20 percent of the features available in the full site. I'd even go further by saying that sometimes, not even the 20 percent of features you take from a parent website should not necessarily be reimplemented "as is." You might want to restructure some of these use-cases and even add new ad hoc use-cases for related scenarios.

A mobile site is a brand-new project that can inherit some code and services from an existing site—more often than not, it inherits large shares of the business logic. This chapter covers a number

of issues and open points that you will want to solve before embarking on building a mobile site. After addressing all these points, building the mobile site is reduced to the work of writing a relatively simple and small website.

From Web to Mobile

You rarely build a mobile site without also having a full site in place. Most of the time, you build a mobile site to serve mobile users better. In some cases, you start building both full and mobile sites at the same time. Currently, it's quite unlikely that you build a mobile site as a stand-alone project. Whatever the case may be, however, this section aims at isolating the issues that differentiate a mobile site from a full website.

If you're building a mobile site as a spin-off of an existing website, then the chances are good that you will be able to reuse large portions of the existing application's back-end code. Those include the data access layer, the domain layer, and possibly a bunch of other distinct components (such as services and workflows) that you may already have in place, which will provide bits and pieces of business logic. If you can replicate some views without too much modification, you may even end up being able to reuse webpages and ASP.NET Model-View-Controller (MVC) controllers—more generally, parts of your presentation and application layers.

As you move towards the presentation layer, though, the chances of reuse diminish. How would you deal with scripts, images, style sheets, and markup? For images and style sheets, there's probably an easy answer—you just have to reduce their size. For scripts and markup, the answer is less obvious and likely is influenced by context.

Application Structure

Before the community of web developers (re)discovered AJAX a few years ago, websites were always built as a navigable collection of distinct pages. Jumping from one page to the next required links and forms. The browser handled each request, which resulted in a full replacement of the current page.

AJAX changed the course of things by using the services of a small browser-hosted component—the *XmlHttpRequest* (XHR) object—through which your script code can play the role of the browser and conduct server requests autonomously. The net effect is that webpages can contain ad hoc script code that use XHR to download data or partial pages. After being downloaded, the content is processed by some other script code and used to update the currently displayed page. AJAX can be used for some specific (perhaps even critical) features, or it can be used extensively throughout the site. When that happens, the site architecture is usually referred to as the *Single-Page Interface* (SPI) model.

The Single-Page Interface Model

In brief, the SPI model refers to a web application that behaves more or less like a desktop application—it has a primary user interface (UI) layout that is adjusted and reconfigured via AJAX calls. The page downloads everything it needs from the source site via script-controlled actions.

The SPI model has been recently formulated as a manifesto. You can read about it here: *http://itsnat.sourceforge.net/php/spim/spi_manifesto_en.php*.

As an early AJAX adopter myself, I've always been a fan of the SPI model, but I've also always seen it as a vector rather than a concrete pattern to implement. As a result, I have not fully implemented the SPI model in a production site yet, primarily due to lack of confidence, proper facilities, and tools. Moreover, I always found it difficult to sell a 100-percent JavaScript site to a customer—at least, that was true up until two or three years ago. Today, the situation is different. The SPI manifesto dates back to the summer of 2011.

With that said, I'm not completely sure that a SPI model is appropriate for just any mobile site. An SPI model requires a lot of JavaScript, partly written by you and for the most part imported from external libraries. You likely need quite a few of these libraries to provide for generic UI manipulation, templates, and data binding. Some of these libraries are based on jQuery and jQuery Mobile. These two libraries alone total some 200 KB (uncompressed) of script and style sheets.

Important In a production site, you typically apply minification and GZIP compression to script and other resources, thus reducing significantly the size of the download. *Minification* is the process of removing unnecessary characters from source code without breaking any functionality. Applied to script files, minification also adds a (thin) layer of obfuscation to your code, making it much harder for humans to read. GZIP is, instead, a popular compression format. Once properly minified and gzipped for a production site, the jQuery library is 31 KB and a bit less for jQuery Mobile.

In addition to script, SPI requires helpers for data binding and UI refresh, which adds a few more tens of kilobytes. Popular libraries in this segment include JsRender, JsViews, Knockout, and Upshot.

In summary, the size of the JavaScript assets that a client will need to download can consume a few hundred kilobytes, which can become a problem. Once downloaded, of course, the browsers will cache the scripts; they don't download the code over and over again. But the browser needs to do a lot of work to render SPI pages—work that goes far beyond simply requesting and rendering markups. The more advanced the device browser is, the more the SPI model becomes affordable (from a resource standpoint) for mobile sites.

Personally, I wouldn't adopt the SPI model without first performing deep analysis of the context. The main challenges with an SPI implementation can be summarized as follows:

- In case of intermittent connectivity, it's difficult to figure out whether the problem is with the application or the network. This may be frustrating. Not that this same problem doesn't exist for other models (i.e., the full-page refresh model), but at least tiny, non-SPI pages download well, even with limited bandwidth. Moreover, with no connectivity at all, you immediately grasp the nature of the problem—when nothing shows up, the problem is not the application.
- Many sites require users to log on before they will serve content. If the users' session expires, the full-page refresh model redirects them to the logon page at the first successive access. The

problem is both manifest and easily resolved. With an SPI site, although user authentication is also AJAX-based, it may take hours before you figure out the root cause of the misbehavior you're observing. In SPI, user authentication requires due attention and effective client and server-side implementation to work smoothly.

So let's explore some other available options.

Note ASP.NET MVC 4 ships with a new project template that promotes the use of the SPI model. The template uses Knockout and Upshot.

Full Page Refresh

At the extreme opposite of SPI lies the classic *Full Page Refresh* (FPR) model. FPR is how the web worked for years: the browser makes a request for a new page, the web server returns the new page, and the browser replaces the current page with the new rendered content. This process occurs repeatedly for each user action.

AJAX (on which SPI is heavily based) made the classic FPR web experience much less compelling, and it also made it more natural for users to expect that only fragments of the current page would be updated as the result of a given action. In a desktop scenario, the FPR model is cumbersome, so more and more large sites are progressively adding AJAX-based capabilities.

However, the significant impact that FPR has had on desktop sites (which have large displays and numerous auxiliary resources for downloading, caching, and refreshing content), is less so on mobile sites because mobile pages are considerably smaller and lighter to begin with. Still, loading several small pages may still be less engaging than updating bits and pieces of the currently displayed page. Updating the current page may still be slow over a slow connection, but at least it doesn't have a dramatic impact on the user experience.

Partial Page Refresh

Some middle ground between SPI and FPR can be found in the partial rendering that both ASP.NET Web Forms and ASP.NET MVC support. In terms of traffic, partial page refresh (PPR) falls between the two extremes—it is not as efficient as SPI, but it's not as poor a user experience as FPR. The idea is that the browser places a request as usual, but that request is captured at the script level—via embedded script—and silently transformed into an AJAX request.

On the server side, the web server handles requests as if they were regular full page requests, but the response is packaged as an HTML fragment. The data being transferred is a mix of data and markup—just the delta between the markup of the current page and the requested new page.

The benefit of the PPR model is that it offers many of the benefits of AJAX without the cost of having to learn new programming features. In ASP.NET Web Forms, PPR happens relatively automatically through the *UpdatePanel* control; in ASP.NET MVC, it occurs through the *Ajax.BeginForm* HTML helper.

In a nutshell, using PPR means that an FPR approach won't require special skills on the development side. In contrast, an SPI model may represent a complete paradigm shift for many developers. PPR represents some middle ground.

Context Is King

Which of the previous options is preferable? There's no obvious and clear answer. If you're determined to find just one answer, then the best approach depends strictly on the context and the knowledge that you have about the devices that are going to access your site. If smartphone traffic is predominant, you definitely can opt for SPI. But if you're interested in reaching the widest possible audience, then you probably should opt for FPR.

The point, though, is that there's probably no ideal solution that works in all cases. Mobile devices are so different that you really should consider partitioning your audience into a few distinct classes of devices and arrange a different solution for each. That could mean serving plain HTML pages to older browsers while implementing a nicer SPI solution for smartphones.

Note In general, you always should consider AJAX seriously because it reduces the amount of network traffic. Extensive use of AJAX, however, actually may *raise* the number of Hypertext Transfer Protocol (HTTP) requests. In a mobile scenario, an HTTP request is more expensive than in a desktop scenario because connections are slower, limited in number (reduced parallelism in download), and also sometimes processed in a more convoluted way, especially if the connection doesn't happen over a WiFi network.

Amount of JavaScript

The amount of JavaScript that you might want to use for the pages of your mobile site is another huge point. Processing JavaScript is crucial to minimize web traffic; on the other hand, it also affects performance. Like it or not, mobile devices (even high-end smartphones) are not as powerful as laptops. Among other things, this means that a mobile device may not be able to tolerate effectively the same amount of JavaScript that you could employ in a full-fledged website. In this context, "not able to tolerate" just means consuming more battery power even if the perceived page performance is acceptable.

More Info The consumption of battery power tends to increase with the number of HTTP requests. Subsequently, a JavaScript-intensive page is critical from the resource management perspective. Here's a reference and some numbers: *http://goo.gl/EKcyj*. Other excellent resources for making sense of the amount of JavaScript and its performance are Steve Souders's blog (*http://stevesouders.com*) and *http://goo.gl/jyhV*.

The jQuery Family of Libraries

Today, the jQuery library is a must for nearly any website. I'm personally dreaming of the day when jQuery will be native to browsers and be integrated into every browser's JavaScript engine. Until that day comes (if ever), jQuery must be linked and downloaded if you plan to use it.

Note that jQuery is required even if you plan to use only jQuery Mobile, which offers a variety of ready-made components for scaffolding a mobile site. Plug-ins for jQuery Mobile can put a "skin" on your mobile site so that it looks like a native application with animations, navigation effects, and snazzy presentation.

All in all, once minified and gzipped, all this JavaScript is likely affordable for use with high-end mobile browsers, even though JavaScript-based effects may emulate some native effects closely but are never the same in terms of performance. With that said, you'll need to be able to limit the quantity of JavaScript in your pages if you want to enlarge your mobile horizons to non-smartphone devices.

So except for smartphones, consider dropping jQuery and derived libraries entirely. On these non–high-end devices, the chances of encountering quirks, bugs, unsupported features, and unexpected behavior is quite high; therefore, why take the risk? By simplifying your page structure, you still should be able to include some JavaScript-based dynamic behavior that relies only on Document Object Model (DOM) support and basic language tools. A good rule of thumb is that (with the notable exception of smartphones and tablets) any quantity of JavaScript beyond 10 KB can begin to degrade load time and performance.

Note Whether you use jQuery or not, *unobtrusive* JavaScript always should be your guiding star. Unobtrusive JavaScript means having a place at the start of the code where you attach handlers to elements so that degradation in the case of unsupported features is easier to handle.

JavaScript Microframeworks

For a desktop site, nobody really minds having a few hundreds of kilobytes in script code. Gmail, for example, fully loaded, usually exceeds the range of kilobytes for loaded JavaScript code. The idea of loading a full-fledged, monolithic JavaScript framework in a mobile site is often unaffordable, especially if you want to target more than just the top iPhone and top Android devices. Still, it's useful to be able to use the services of existing code and ready-made frameworks. JavaScript microframeworks come to the rescue.

Microframeworks are small, highly focused libraries whose overall size is often only a few kilobytes; sometimes even 1 KB or so. There's no magic and no special tricks—microframeworks are so small not only because they are minified and gzipped, but also because they take on just one or two tasks.

You probably will want to pick up a library for asynchronous (async) loading: one for optimized DOM traversing, one for touch gestures, and perhaps a few more. You can find an interesting list of such microframeworks at *http://microjs.com;* their average size is around 1 KB.

One microframework for general-purpose JavaScript programming in the context of mobile sites is XUI (see *http://xuijs.com*). XUI is not specific to a single task as are some of the libraries listed on *microjs.com*; instead, it's specifically designed for mobile scenarios, so you can consider it a competitor to other larger mobile frameworks such as jQuery Mobile and Sencha Touch. Unlike these larger frameworks, XUI doesn't force you into a given programming paradigm or page structure. It focuses on a few tasks (e.g., DOM/CSS management) and totals only 5 KB gzipped. XUI is a good alternative to jQuery libraries for mobile sites and offers a powerful alternative to jQuery libraries for lower-end devices that support JavaScript, DOM, and Cascading Style Sheets (CSS).

Chapter 5, "HTML5 and jQuery Mobile," will cover the whole topic of jQuery Mobile. Sencha Touch is a JavaScript framework, originally created to add touch capabilities to mobile pages, but which is now a full-fledged framework for developing mobile web applications that look and feel native on most mobile platforms such as iOS and Android. You can find out more about SenchaTouch at *http://www.sencha.com*.

More Info A good resource for microframeworks, and for comparing their pros and cons against larger script frameworks such as jQuery, is Addy Osmani's work at *http://goo.gl/jnE9t*.

Application Device Profiles

Device fragmentation is huge in the mobile space. If the differences between browsers in desktop site development scare you, then be aware that the mobile space is much worse.

On rare occasions, you can get by with just one set of pages for a mobile site. Ideally, you need pages that can adjust intelligently to the characteristics of the requesting browsers. Sometimes you just end up having multiple versions of the same page—one for each device (or class of device) that you intend to support. Alternatively, sometimes you may have one common page template that you fill up with device-specific content. But at the end of the day, these are implementation details.

The crucial point is that you need to split your expected mobile audience into a few classes. Then, for each page or view, you provide class-specific markup. This is the essence of *multiserving,* as briefly described in Chapter 3, "Mobile Architecture." Chapter 6, "Developing Responsive Mobile Sites," will illustrate multiserving with an example.

Note In most developed markets today, it is possible to cover most devices with good semantic markup of some kind and then progressively enhance the presentation and interaction using CSS and JavaScript. In this regard, the concept of device classes applies to CSS as well: however, if you want to present different content to different classes of devices, then you need to use multiserving because the markup will change between classes.

Practical Rules to Class Profiles

In the mobile space, neither the team building a given site nor the team producing a general-purpose library can afford optimizing pages on a per-device basis—the number of potential devices to take into account is just too large (in the order of several thousand). Hence, a common practice has become to classify all the devices you're interested in into a few classes. How many classes do you need to have, and how do you define a class?

A device class typically gathers all devices that match a given set of capabilities. You define the classes based on the business cases and scenarios. A basic (but still arbitrary) classification may consist in splitting the whole range of requesting devices into three categories: smartphones, tablets, and all other browsers. You provide a rich web experience to smartphones, serve the full site to tablets, and offer plain HTML to all others.

How would you define a smartphone? Everyone would agree that an iPhone is a smartphone while, say, a Nokia 7110 is not. [The Nokia 7110, released in the fall of 1999, was the first device equipped with a Wireless Access Protocol (WAP) browser.] In contrast, the Nokia 6600 certainly was a smartphone when it came out in 2004, but nobody would consider it a high-end phone today. But there are no hard and fast rules. You should know that not only the device classes, but also the rules that determine which class a given device belongs to, are highly variable and strictly specific to a business scenario. At the same time, this lack of fixed rules and practices makes it possible for everybody to define classes in a way that best fits a given business. All you need is a reliable infrastructure that first identifies the device and then tells you about its real capabilities. I'll return to that infrastructure in a moment.

As a purely intellectual exercise, here are some requirements that identify a modern smartphone in 2012. Note that these are likely to change, even in the near future.

- *Operating system (minimum version)*: all versions of iOS, Android 2.2, Windows Phone 7, RIM OS 6, Samsung Bada 2.0, Symbian Anna, and Nokia Meego
- *Input mode*: touchscreen
- *Screen size*: 320 × 480 pixels

Admittedly, this definition is rather arbitrary, and it may sound too restrictive for some while being way too relaxed for others. Above all, this definition will become progressively less pertinent as more powerful devices hit the market.

A tablet has two main characteristics—it is a mobile device (not a desktop browser), and it has a larger screen than a smartphone. You can probably set the lower boundary to 640 pixels today.

It is important to note that grouping all the remaining devices into a single class may be a tough decision. Whether you really want to serve plain static HTML to all of them or further split the remaining devices into two or more classes is up to you.

How can you discover the capabilities of the browser for each device? How can you be sure that the requesting browser is really a mobile device? If it's sufficient for your needs to just check the screen size and screen resolution, then you might decide to go with CSS media queries for high-end

browsers, and use script code that simulates that on older browsers. Although this approach may work in some cases, it's not a route that will take you far. Instead, relying on a commercial device description repository (DDR) is probably the best way to go. Chapter 6 discusses a few DDR frameworks, focusing in particular on WURFL.

Dealing with Older Browsers

When I talk to executives planning the mobile strategy of their companies, I often get the impression that when they say "mobile," they just mean the iPhone and iPad. While it can't be denied that iPhones and other smartphones are actually responsible for most of the mobile traffic to sites, the mobile universe contains many other types of cell phones and devices as well. Unfortunately, not all of them have the same characteristics, but they are so numerous that you must find a common denominator approach.

In the process that you use to identify the device profiles to support, you must define the bottom of the stack at some point. Devices that fall into this sort of catchall group typically are served plain HTML or, at least, the simplest markup that your mobile site can serve.

What's the best way to handle this?

When you end up having a catchall device profile, it also means that there's some rich library you're relying on for higher-end devices. The jQuery Mobile library is an excellent example. Such mobile libraries sometimes offer to scale down the otherwise rich markup they produce on older browsers automatically.

That's apparently a fantastic deal for you: you write the mobile markup once, and it gets downgraded automatically for less powerful browsers. Unfortunately, my current experience has not been particularly positive on this point. Although most libraries do fulfill their promises of downgrading the markup, the quality of the HTML that they produce when that happens is often below your desired standards. You probably want to take care of the HTML being served to older devices yourself rather than blindly relying on the kind of hard-coded markup served by some libraries.

The bottom line is that while jQuery Mobile (and other libraries) can truly downgrade HTML based on the requesting browsers, you'll achieve a better final effect if you manually fix up the output. Currently, I'm inclined to use jQuery Mobile—but only to serve smartphones and tablets.

Note Chapter 5 will cover jQuery Mobile in more detail, including more about its browser-graded support matrix. That feature is orthogonal to performing your own device capability detection, but overall, I prefer to skip the automatic downgrading, at least with the current version of most libraries.

Optimizing the Payload

Minimizing the number of HTTP requests to websites is always a good thing, and it should be a central aspect of any strategy aimed at improving the performance of a site.

If you have ever tried to use a mobile device to connect to a very basic site with a few plain HTML pages, a bit of CSS, and one or two images over a 3G data connection, you have experienced a delay, or *latency*. This latency is relevant if the device is not one of the latest smartphones with a powerful processor. In the mobile space, minimizing the total amount of data transferred and the number of requests is not simply a matter of optimization; it is a crucial development point.

Over the years, a number of recommended practices have been worked out to help developers build fast websites. Yahoo! has been quite active in this field, publishing a very valuable document that you can read here: *http://developer.yahoo.com/performance/rules.html*. The rules in that document are written for a generic website, but for the most part, they can be applied equally well to both desktop and mobile sites. Here's a summary of the key suggestions:

- Take care of the page structure and find the right place for scripts and style sheets.
- Reduce the number of HTTP requests.
- Reduce the size of resources.
- Maximize the use of the browser cache.

Let's briefly go through the optimization aspects that are most relevant to mobile sites next.

The Page Structure

A mobile browser may load pages significantly slower than a laptop browser. This means that not just raw performance but also *perceived* performance is important. Little tricks, such as placing style sheets at the top of the page in the <*head*> section help, because doing that means that the body of the page will be ready to render as it is downloaded. The overall download time doesn't change, but at least users see some results a bit sooner.

Similarly, placing scripts at the bottom of the page is helpful because it reduces the impact that synchronously downloading scripts may have on page rendering. When browsers encounter a <*script*> tag, they stop page rendering and proceed to download the script synchronously. Page rendering resumes only after the script files have been downloaded, parsed, and executed. When all the scripts are at the bottom of the page, browsers don't need to interrupt the page rendering process to load scripts; therefore, browsers are free to concentrate on displaying an early view of the page.

Reduce the Number of Requests

Too many HTTP requests are the primary cause of latency in websites. This statement is even truer for mobile sites. Executing an HTTP request is an expensive operation, especially when that entails connecting to a radio cell. In this case, to preserve battery power, the device sometimes cuts off the connection right after receiving the HTTP response, meaning that to execute another request, a new handshake is required, which consumes both time and resources.

For this reason, it is doubly sinful to let links to duplicate resources go unnoticed. Linking a script twice doesn't affect the rendering of the page, but while the performance hit is negligible on a laptop, it becomes a serious performance hit in a mobile scenario.

The same can be said for redirects. Each redirect requires two HTTP requests. Avoiding redirects from a site is one way to reduce the number of requests that devices visiting that site have to place. Most ASP.NET MVC books [including my book *Programming ASP.NET MVC*, 2nd ed. (Microsoft Press, 2011)] recommend that the Post-Redirect-Get pattern is appropriate for input forms because it saves applications from unwanted F5 refreshes. In a mobile space, that also introduces extra requests; make sure that you make a sound decision about your projects on this point. If you can afford to use AJAX, that would probably be an ideal compromise.

Compacting Resources

When your goal is to reduce the number of HTTP requests, there's little better than merging two or more files. Image sprites, for example, illustrate just this point. A *sprite* is an image that results from the concatenation of two or more images. That way, the browser can make a single request, downloading and caching multiple images at one time.

Image sprites are not always ideal. Using sprites works great with very small images, such as button icons that are widely used across the pages, but sprites may not be as ideal for the relatively large images used by distinct pages. Downloading a 50 KB image may not be easy over a 3G connection and with an older browser, so if the image is part of a sprite and the sprite size is 100 KB or more, downloading it would take even more resources—probably enough to make the user experience unpalatable.

Sometimes, to save an HTTP request, you can decide to encode a small image (one on the order of just a few kilobytes) as a Base64 string and embed it directly in the page. The data Uniform Resource Identifier (URI) scheme just serves this purpose.

The data URI scheme defines a standard for embedding data within the page instead of linking it as an external resource. The scheme is defined in RFC 2397; you can read about this RFC on Wikipedia at *http://en.wikipedia.org/wiki/Request_for_Comments*. The net effect of applying the data URI scheme is that the content of the *src* attribute of the *<img>* element matches the following template:

```
data:[<content-type>][;base64],<bytes of the image>
```

First, you place the data keyword followed by the Multipurpose Internet Mail Extensions (MIME) type of the image. Next, you place the Base64 keyword followed by the Base64-encoded representation of the binary image. If you were embedding an image manually, here's the markup you would need:

```
<img class="image" src="..." alt="" />
```

The Base64 image encoding (also known as *image inlining*) saves a few HTTP requests and improves both the *real* performance and *perceived* performance of the page. It is not a feature to use for just any image and page!

Note Most modern device browsers support image inlining, with the sole notable exception of the Windows Internet Explorer browser embedded in the versions of Windows Phone prior to version 7.5. However, few older browsers support this feature, whose importance decreases as the capabilities and processing power of the browser increase.

Improve Your Control over the Browser Cache

If the primary objective of a mobile site is to minimize the number of requests, browser caching is the primary tool that you have to manage. Moving auxiliary resources (i.e., style sheets and scripts) to external files often helps. Initially, the number of HTTP requests is higher, but after the first access auxiliary resources are cached, no more requested expiration occurs.

External resources are really beneficial if such resources are widely referenced from a variety of pages. In home pages and rarely visited pages, inlining of resources (where possible) may be a better option. In addition, you can drive the browser behavior about caching by using e-tags and *Expires* headers on your critical resources.

Note In addition to optimizing the cache and minimizing HTTP requests, you might want to employ all possible techniques that reduce the amount of data to download. This certainly includes fixes to application logic to return the smallest possible amount of data and markup, but it also includes actions on the infrastructure, such as enabling compression at the web server level and minifying scripts and style sheets. Finally, note that you should always return data as JavaScript Object Notation (JSON) strings rather than XML strings. (JSON was once named the "fat-free alternative" to XML.)

The browser cache also applies to AJAX responses. The use of AJAX makes the request go unobtrusively for the user, who remains in total control of a still responsive page. However, it doesn't mean that the request will end in a matter of milliseconds. Sometimes caching the AJAX response helps to make most responses really instantaneous. This pattern, however, doesn't work for all types of requests. If you place a request to read the current balance of an account, you don't want it to be cached. If you use an AJAX request to auto-complete a text field, you want to cache as much as possible. This is to say that control over the AJAX cache must exist, and in real-world scenarios, it must be applied on a per-URL basis. Most libraries, though, offer an all-or-nothing control, which is hardly the right thing for a mobile site.

Important As the Back button is the most popular button in a browser, you need to remember this point: you don't want the page to reload when you hit that button. To avoid reloading, it is not always enough to set the right cache headers. The size of the elements also matters. The following link provides some numbers. It is a post from a couple years ago, but it is still worth reading: *http://www.stevesouders.com/blog/2010/07/12/mobile-cache-file-sizes*.

The Offline Scenario

A mobile site represents an excellent shortcut to implementing a mobile strategy because it brings products and content to a variety of devices without writing a different application for each mobile platform that you intend to support. At the same time, a mobile site requires constant connectivity to work well. This aspect of mobile sites is going to be more and more of a showstopper for sites, and it highlights the key difference between mobile sites and native applications. It's not coincidental that offline applications have been given a role in using HTML5.

Offline Sites with HTML5

An offline experience without an HTML5-enabled browser is quite hard to achieve; while it's not impossible, it does require a strong commitment. In other words, it's not a feature that you would want to offer as a free add-on to a customer!

When most people talk about offline sites, they mostly mean online sites that perhaps occasionally experience long downtime periods. The key to surviving such lack of connectivity is caching—in particular, to take advantage of the browser's ability to cache resources that a user may navigate to later, during a downtime window. These resources include not only auxiliary files, but also AJAX responses.

HTML5 lets you create a manifest file and link to it from the *<html>* tag of the home page. In this manifest code, you list the files you want to keep cached, which resources are a fallback for other (possibly missing) resources, and which resources are available only while online:

```
<!DOCTYPE html>
<html manifest="/offline.appcache">
  ...
</html>
```

The browser's ability to cache a subset of the full site on the client isn't a great thing, per se. It requires more than that to be truly effective. That means is that there is little value for users in just navigating through a few static pages. While that's considerably better than getting a 404 error message from the browser, it's not really a decisive change.

Persisting Application Data

Persisting application data locally is another aspect of websites strictly related to surviving an offline status interval. In an HTML5-enabled browser (again, most browsers in today's smartphones are HTML5-ready), you use the local storage application programming interface (API) to write name/value pairs in an application-restricted area managed by the local browser. In the near future, the flat name/value format may become even more sophisticated and evolve into a table-based and indexed format.

Persistence is also related to synchronization. The ability to persist data locally fully enables occasionally connected application scenarios. At the same time, in an occasionally connected scenario, you might want to offer a read-only view of the data or enable updates. When this happens, you have either the problem of queuing operations or the problem of editing a local cache of data to be synced with the server when the connection is re-established.

Overall, if you want to build a full-fledged, occasionally connected scenario, you'd better endow yourself with a solid sync framework and/or consider using the OData protocol and facilities for your data exchange.

Development Aspects of a Mobile Site

Based on the discussion in Chapter 3, the most important task when planning a mobile site planning is selecting use-cases. This doesn't mean, however, that use-case selection is unimportant when developing full sites or other types of applications. It's just that a mobile application and site are structurally built around a few (and well-chosen) use-cases.

Even when you're simply picking up a use-case from the root site, the way in which you implement it for a mobile audience may require significant changes—possibly a different user interface and perhaps even a different workflow.

At this point, let's assume that you have a well-defined (and hopefully well-chosen) set of use-cases; and let's also suppose that you have everything required for the back end already in place. You are now ready to start producing markup. But, first and foremost, how do you reach the mobile site?

Reaching the Mobile Site

A mobile site can be a stand-alone site located at its own unique URL, or it can result from the application logic serving appropriate content to desktop and mobile browsers. In the former case, you just create a new ASP.NET project, design your pages for the mobile devices that you intend to support, give your images and style sheet the size and properties they need, and go. In the latter case, you have a single project where you just handle the desktop full-web case as a special case of a mobile site. Let's investigate the two options.

One Site, One Experience

Maybe partly influenced by the One Web vision, I initially approached mobile development with the idea of offering my users just one endpoint and host name. The plan was to hard-code the server with the ability to detect device capabilities and serve the most appropriate content. So my first mobile project was a mere extension of an existing site: I just released a new version of the desktop site with the additional ability to detect mobile browsers and serve ad hoc markup. I had just one ASP.NET project with two distinct sets of pages/views for desktop and mobile.

Honestly, this didn't take much effort to design and implement. It took only a little extra engineering to set up a page/view router to distinguish between and serve both mobile and desktop requests. However, testing the site was painful. The only reliable source of information was to use a real mobile device; switching the user agent string on desktop browsers just didn't work effectively. We'll return to these test issues later in this chapter, and also demonstrate this one-site, one-experience approach while presenting a demonstration of device capability detection.

The noteworthy point is that when you have just one site that can handle both mobile and desktop browsers, you actually have one set of pages for full browsers and then multiple sets of pages for each class of mobile devices that you support. Really, the desktop becomes the special case!

What about different use-cases? This isn't a big issue. You always have a home page in both mobile and desktop environments, so your mobile home page will just offer a different set of links and start a different type of navigation. You only need some logic that, when a new browser session starts, the home page request for *http://www.yoursite.com* produces the output of *default.aspx*, *default.mobile.aspx,* or perhaps *default.iphone.aspx*.

Two Sites, One Experience

Mostly for ease of development and testing, I soon switched to a different model: two sites, one experience. The mobile site is a neatly separated entity and has its own URL. You have a stand-alone mobile site reachable as a */mobile* virtual directory of the main site or as a subdomain, such as *http://m.yoursite.com*. Sometimes, the site takes its own extension, such as *http.//www.yoursite.mobi*. Any option that you choose here is equally good, in my opinion, and mostly depends on other aspects of the mobile strategy. In any case, it is always a good advice to provide users (at least on smartphones and tablets) with a link to browse the full site.

Because the mobile site is isolated from the principal site, you can test it much more easily—and you can use desktop browsers or mobile generic emulators (such as Opera Mobile Emulator) to perform quick tests aimed primarily at evaluating the user interface and experience. Obviously, it's crucial to test on real devices, but for quick tests on markup, colors, position, and flow, using a desktop program makes the process seamless.

Because you now have two distinct sites, you need an automatic mechanism to switch users to the right site based on the capabilities of the requesting device.

Routing Users to the Right Site

It's a mistake to assume a one-to-one correspondence between desktop and mobile pages. This may happen but should not be considered a common occurrence. Note that by saying "page correspondence," I simply mean that both applications can serve the same URL; I'm not saying anything about what each page actually will serve.

All in all, we can safely consider only the host name of any requested URL. If the host name belongs to the desktop site and the requesting browser is detected to be a desktop browser, then everything works as expected. Otherwise, the user should be displayed a landing page, where she will be informed that she's trying to access a desktop site with a mobile device. The user is given a chance to save her preference for future similar situations. The preference is stored to a cookie and checked next.

If the request refers to a URL in the mobile site and the user seems to have a desktop browser, consider showing another landing page rather than simply letting the request occur as usual. Finally, if a request is placed from a mobile device to the mobile site, it will be served as expected; namely, by looking into the device capabilities and figuring the most appropriate view. Figure 4-1 presents a diagram of the algorithm.

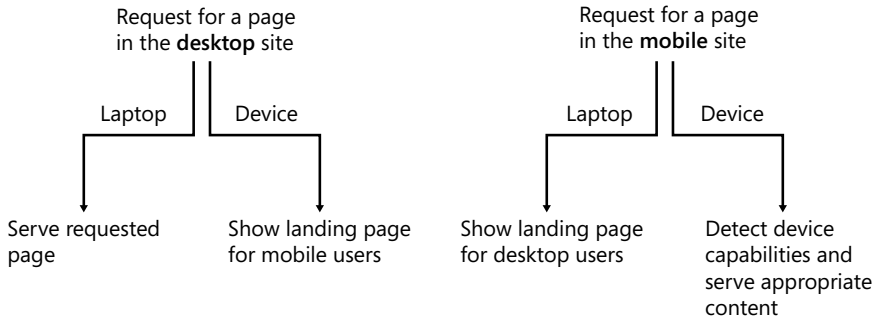


FIGURE 4-1 The desktop/mobile view switcher algorithm.

How would you implement this algorithm?

In ASP.NET, the natural tool to implement this routing algorithm is an HTTP module that is active on both sites and capturing the *BeginRequest* event. The module will use plain redirection or, if possible, URL rewriting to change the target page as appropriate.

Here's some code that implements the aforementioned algorithm in the desktop site:

```
public class MobileRouter : IHttpModule
{
    private const String FullSiteModeCookie = "FullSiteMode";
    public void Dispose()
    {
    }
    public void Init(HttpApplication context)
    {
        context.BeginRequest += OnBeginRequest;
    }

    private static void OnBeginRequest(Object sender, EventArgs e)
    {
        var app = sender as HttpApplication;
        if (app == null)
            throw new ArgumentNullException("sender");

        var isMobileDevice = IsRequestingBrowserMobile(app);

        // Mobile on desktop site, but FULL-SITE flag on the query string
        if (isMobileDevice && HasFullSiteFlag(app))
        {
            app.Response.AppendCookie(new HttpCookie(FullSiteModeCookie));
            return;
        }

        // Mobile on desktop site, but FULL-SITE cookie
        if (isMobileDevice && HasFullSiteCookie(app))
            return;

        // Mobile on desktop site => landing page
        if (isMobileDevice)
```

```
            ToMobileLandingPage(app);
    }

    #region Helpers
    private static Boolean IsRequestingBrowserMobile(HttpApplication app)
    {
        return app.Context.Request.IsMobileDevice();
    }

    private static Boolean HasFullSiteFlag(HttpApplication app)
    {
        var fullSiteFlag = app.Context.Request.QueryString["m"];
        if (!String.IsNullOrEmpty(fullSiteFlag))
            return String.Equals(fullSiteFlag, "f";
        return false;
    }

    private static Boolean HasFullSiteCookie(HttpApplication app)
    {
        var cookie = app.Context.Request.Cookies[FullSiteModeCookie];
        return cookie != null;
    }

    private static void ToMobileLandingPage(HttpApplication app)
    {
        var landingPage = ConfigurationManager.AppSettings["MobileLandingPage"];
        if (!String.IsNullOrEmpty(landingPage))
            app.Context.Response.Redirect(landingPage);
    }
    #endregion
}
```

Once installed in the desktop site, the HTTP module captures every request and checks the requesting browser. If the browser runs within a mobile device, the module redirects to the specified landing page. The landing page will be a mobile optimized page that basically offers a couple of links: one to the home of the desktop site and one to the home of the mobile site. Figure 4-2 shows a sample landing page viewed with an Android 2.2 device.

If the user insists on viewing the full site, then you can't simply redirect to the plain home page. For its nature, the HTTP module will intercept the new request and redirect again to the mobile landing page. From the landing page, you can simply add a specific query string parameter that the HTTP module will detect on the successive request. Here's the actual link that results in Figure 4-2:

```
<a href="http://www.easycourt.net/contosoen?m=f">Full site</a>
```

You are responsible for defining the query string syntax; in this case, *m* stands for mode and *f* for full. The task is not finished yet, though. At this point, users navigate to the home page of the site. What about any other requests? Those requests, in fact, will be intercepted by the HTTP module. By adding a cookie, you can provide additional information to the HTTP module about requests deliberately sent to the desktop site from a mobile device.

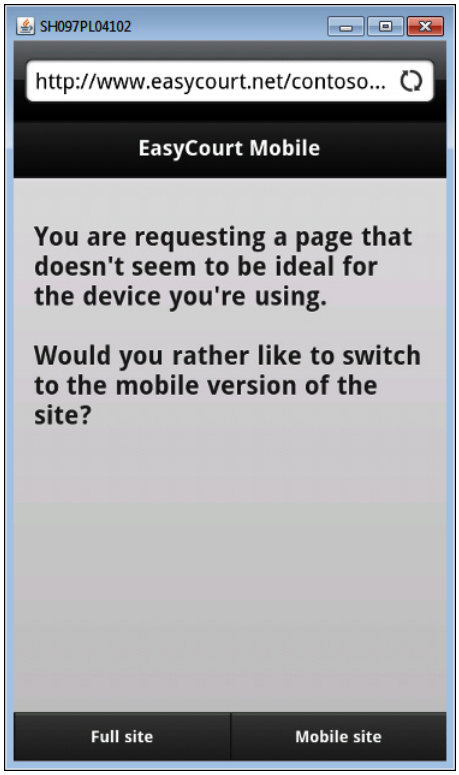


FIGURE 4-2 The landing page of the EasyCourt demo site.

How can the user switch back to the mobile site? Ideally, any desktop site with a sister mobile site should offer a clearly visible link to switch to the mobile version (and vice versa when the full site is viewed on a mobile device). If not, the user won't be offered a chance to choose the full or mobile site until the cookie expires or is cleared. To clear cookies, users deal with the Settings page of the mobile browser.

Adding Mobile Support to an Existing Site

Where do you place the landing page? Is it on the desktop or on the mobile site? In general, it doesn't matter; however, if you put it on the mobile site, then you really can enable a scenario in which you deploy a mobile site with all the required routing logic without touching codebase of the existing desktop site.

In the demo site of EasyCourt, a commercial booking system for tennis courts, which I introduced in Chapter 3, I just edited the Web.config file of the desktop site and deployed a library with the HTTP module in the Bin folder. No changes were made to the source code. Here's the configuration script to add a router HTTP module to the desktop site:

```
<system.webServer>
     <modules>
         <add name="MobileRouter" type="..." />
```

```
    </modules>
    ...
</system.webServer>
```

The Welcome page was defined on the mobile site. Note that the Welcome page always should be visible, and it never should need authentication. Depending on how you deploy the mobile site—a distinct root site/application or a child application/directory—you may need to tweak the Web.config file of the mobile site to disable the HTTP module. If the mobile site is a distinct application, then it needs its own Web.config file that has been fully configured with the HTTP module. If the mobile site is, instead, hosted as a child directory in the desktop site, then it inherits the configuration settings of the parent site (the desktop site), including the HTTP module. To speed up requests, you might want to disable the HTTP module in the mobile site.

Here's the configuration script that you need in the mobile site's Web.config file. The script clears the list of HTTP modules required by the mobile site:

```
<system.webServer>
    <modules>
        <clear />
    </modules>
    ...
</system.webServer>
```

In addition, you need to instruct the parent application/site explicitly to stop the default inheritance chain of settings. Here's what you need:

```
<location path="." inheritInChildApplications="false">
<system.webServer>
    <modules>
        <add name="MobileRouter" type="..." />
    </modules>
    ...
</system.webServer>
</location>
```

Also, notice that when the mobile site is a child application/directory, then it inherits a bunch of settings (for the section where inheritance is not disabled) that don't need to be repeated (for example, connection strings and membership providers).

Note This example is based on the default Internet Information Services (IIS) 7.5 configuration—integrated pipeline mode. If you're using the classic pipeline mode, then instead of *system.webServer/modules,* you should operate on the *system.web/httpModules* section.

Design of the Mobile Views

Mobile websites normally show a subset of the content offered by a desktop site. For example, if you have a three-column layout in a desktop site, you probably want to remove (or move to additional pages) content displayed in two of the three columns. Reducing the amount of information improves the load time of the site and makes better use of the available space. "Shrink-and-fit" is a popular slogan.

An ideal layout for pages of a mobile site is based on a single column. The font size is large enough to allow easy reading without zoom. The search bar (if any) ideally will go at the top, and navigation links are commonly placed at the bottom. You might want to place a link to the desktop version of the site somewhere on the mobile site.

Scrolling is accepted, especially vertical scrolling, but endless lists are annoying. Let's briefly review some common scenarios now.

Input Elements

On a typical mobile site, most of the functionality is read-only. This fact seems to suggest that you are not going to have that many chances to write input forms, and perhaps even that input forms are not an aspect of development you want to invest much time in.

This is just wrong.

Exactly because most of the functionality is read-only and a mobile device is tiny and not as powerful as a laptop, providing specific and restricted parameters is key. To type query parameters, or to specify settings, input forms are a common presence on mobile pages anyway. Some typical web and Windows controls need fixes, though, especially in light of the touch capabilities of many devices.

Typing text on mobile devices is hard and should be minimized. As we'll see in later chapters, this is easier to achieve with native applications, where you can control the input scope of soft keyboards and attempt to display an optimized subset of keys to the user. In mobile sites, all is left to the browser, although developers can use forms of auto-completion via AJAX.

If the browser understands HTML5, then you can just use the most appropriate *type* attribute on the <*input*> element and let the browser do the rest. To be honest, what you get may differ quite a bit, even on smartphones. For example, numbers and ranges are well supported on both iPhone and Android at present, but the date type produces just the plain text box on Android. In iOS5, however, the browser recognizes your intentions and displays the compelling iPhone date picker element (see Figure 4-3).

On the other hand, iOS still lacks the ability to upload files from the browser, which is a feature available in Android mobile browsers.

In general, data entry should be redesigned and even rethought case by case. For example, when it comes to booking a tennis court in EasyCourt, the site offers a drop-down list with ready-made dates, as shown in Figure 4-4. It should be noted, however, that this particular screen is not mobile-specific but simply represents the plain transposition of a desktop page. You always should wonder if there may be an innovative way of letting users enter their choices. Don't stop at the classic, desktop-oriented way of building input forms. Mobile is different and user-centered.

FIGURE 4-3 Entering dates in iOS5.

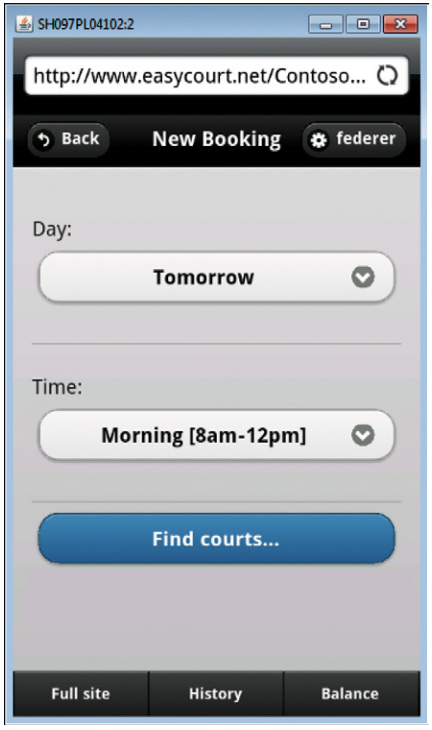


FIGURE 4-4 Setting parameters for a query on available courts.

And finally, here's a quick tip related to passwords. Strong passwords typically require mixing lowercase and uppercase letters and numbers and symbols, but that's too much work in a mobile website. If possible, consider using numeric personal identification numbers (PINs).

Radio Buttons and Check Boxes

A common presence in many input forms, radio buttons and check boxes have just one problem in mobile forms: they tend to be too small and hard to select with a touch. It is recommended that you style these elements accordingly using padding and the companion *label* element. Don't place such elements too close to other elements, including other possibly related buttons.

Note On iOS, the tappable region is 44 points, which corresponds to a square of 7 mm. This should be considered if you are creating a mobile site without using a framework such as jQuery Mobile, which shields you from many details.

It is worth noting that jQuery Mobile completely rewrites the canonical markup for such input elements, as follows:

```
<input type="checkbox" name="rememberme" id="rememberme" data-theme="d" />
<label for="rememberme">Keep me logged</label>
```

The output produced by the preceding markup is shown in Figure 4-5.

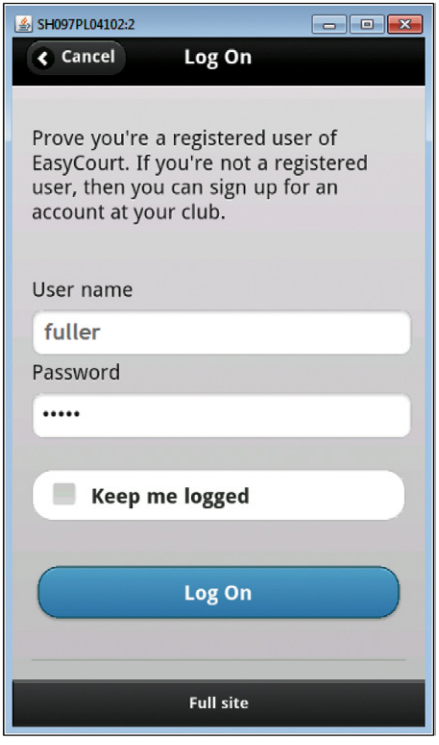


FIGURE 4-5 A check box that looks like a button and is really easy to tap.

In particular, jQuery Mobile surrounds the base HTML markup with padded DIVs to make it easier for the user to tap. Compare the experience of tapping this with a regular check box that may even need zooming to be read and selected (see Figure 4-6).

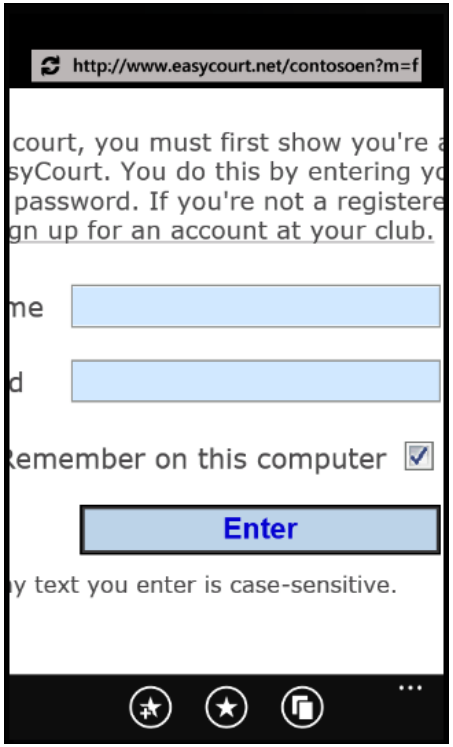


FIGURE 4-6 A regular HTML check box, but zoomed.

When the user's choice is a *yes/no* or *on/off* choice, it is common in mobile to use flip-switch artifacts that are larger and more comfortable to use and easier to see. Although differently styled, flip-switches are available in iOS, Android, and Windows Phone.

Scrollable and Drop-Down Lists

Radio buttons are a valid solution if you need only one selection out of just a few options. When the number of options gets closer to 10, or more, you might want to consider a combo box or a plain scrollable list.

One of the original Windows controls, the combo box provides the same experience on desktop applications and websites. The same points previously discussed for radio buttons and check boxes apply to combo boxes too—they're usually too small to be touch-enabled effectively. You can play with style sheets and use a larger font, but you manage to improve the result only a little bit—it remains far from ideal. Figure 4-7 compares native and adjusted combo boxes.

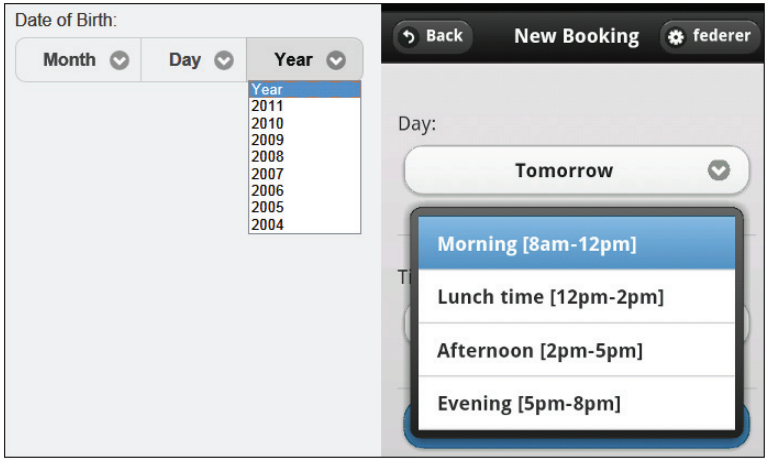


FIGURE 4-7 Native and adjusted combo boxes.

Figure 4-7 shows the results that you get when you use jQuery Mobile. The bottom line, though, is that you hardly want to use plain <*select*> elements on a mobile page without some graphical adjustments for size and touchability. On the other hand, <*select*> elements are more important than ever in mobile pages because they can save users from typing free text.

Scrolling vertically is quite a natural gesture in a mobile scenario. However, the more you can group related items, the better the final experience for the user is. Once again, focus is king. Figure 4-8 shows a couple of mobile views featuring grouped items and collapsible elements that effectively save valuable screen real estate while providing optional information to users. Grids are effectively rendered on mobile only rarely; smartly built vertical lists are a much more common and preferable choice.

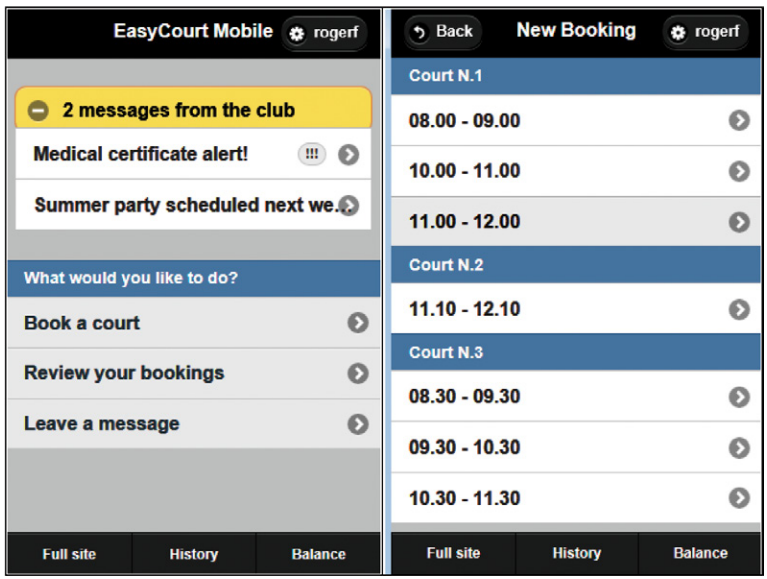


FIGURE 4-8 Collapsible and grouped elements.

In Figure 4-8, a collapsible element is the block that shows messages. As the icon suggests, it is an area of the screen that can be collapsed. Other blocks are tables with one or multiple sections (to use a terminology popular to iOS and Android developers). Some of the items serve as dividers of the sections and are styled differently, communicating the idea of grouped items.

If the list of items is particularly long, you might want to introduce some pagination. Essentially, pagination works by first loading a fixed number of items and then displaying a link at the end to get more. Alternatively, more items can be loaded automatically when the user reaches the bottom of the list. It is up to you to ensure that the DOM doesn't end up containing too many items after a few click requests for more items. If this happens, you might want to remove older items and move them to another page that the user can request if needed.

Note If you want, you can play live with EasyCourt at *http://www.easycourt.net/contosoen*. By reaching the site with a laptop and a mobile device, you can experience the different levels of usability. The desktop site didn't undergo any change to support mobile devices except for the configuration file. For this reason, the desktop site currently doesn't include a link to the mobile site. This means that if you switch to the full site from a mobile one, you can't change back until you remove cookies for the site manually using the browser's Settings page.

Free Text and Auto-Completion

No matter your efforts to save free text typing, sometimes users are just requested to enter a name or a comment. There's not much you can do to reduce the hassle of typing on a mobile soft keyboard if you can't figure out ways to allow the browser to offer a close-enough input scope. With native applications, it's slightly better, but text typing remains a very sore point for mobile developers.

Auto-completion does help, but it costs you quite a bit of script code, which in turn increases the payload for the page. Auto-completion must be coded through a plug-in—the jQuery UI auto-complete plug-in works well with mobile pages. A plug-in that downloads data from an external source is the most viable option when the potential number of options is in the order of hundreds. For a smaller number of items, you can also consider the jQuery Mobile filter bar. The base markup is the following:

```
<ul>
  <li> ... </li>
  <li> ... </li>
  <li> ... </li>
</ul>
```

Additional attributes will give the final markup a strong auto-completion flavor, as in Figure 4-9. Auto-completion also is sometimes a way to avoid long lists of hundreds of items that are very boring to scroll through.

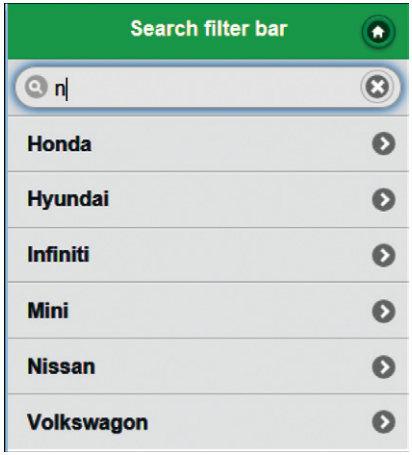


FIGURE 4-9 A jQuery Mobile filter bar selecting list items with "n."

Testing the Mobile Site

Testing a mobile site is not an easy task. You may use a mobile emulator (possibly more than one to better simulate the various scenarios), but emulators are not enough. Testing on real devices is what really matters and gives you the real perception of the application performance and user experience.

However, before you deploy to a few test devices (at least one per device class), you might want to check user interface and logic on a more comfortable desktop machine. Emulators and user-agent switchers can do some of the work for you.

Desktop Emulators

An *emulator* is a desktop application typically running on Windows that aims to mimic the behavior and functions of a specific browser. In mobile development, emulators are time-saving tools for first-pass software testing.

Device/OS emulators are more specific than browser emulators. Device/OS emulators (i.e., the Windows Phone emulator) are created by device manufacturers and are generally really close to the device.

Browser emulators are generally written by third-party companies and serve mostly for verifying the look and feel of the site at a given size and resolution. You should not consider a browser emulator for verifying the rendering of pages that have been optimized for certain classes of devices.

In general, emulators are a first-aid device, and they can be used to test the behavior of the application on types of devices that you have no access to.

User Agent Switching

Most desktop browsers come with tools to switch the user agent (UA) string. By changing the default UA string to that of a mobile device, you allow the browser to present the website with the credentials of a particular mobile browser. Subsequently, the website will serve the mobile content that it has in store. UA switchers are a special case of browser emulators.

The Firefox browser was the first to offer such a powerful add-on. Today, you find the same tool for Google Chrome and Internet Explorer 9, as shown in Figure 4-10.

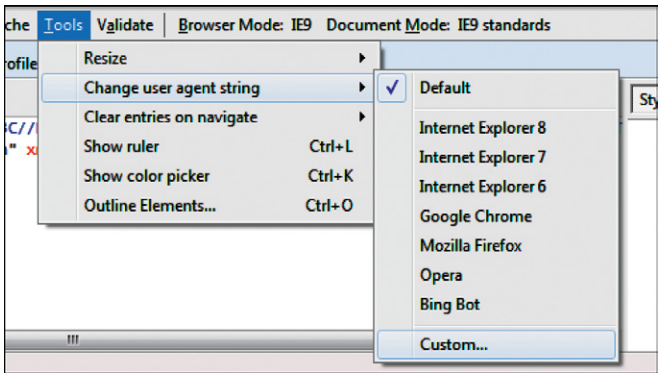


FIGURE 4-10 The User Agent Switcher tool in Internet Explorer 9.

Opera Mobile Emulator is another interesting tool through which you can experience your site as if you are visiting it through a number of specified devices. Compared to the User Agent Switcher Tool, the Opera Mobile Emulator is more flexible; more important, it is a single tool that can serve multiple scenarios. In Figure 4-11, you see the emulator configured to give the site being tested the appearance of running on a tablet.

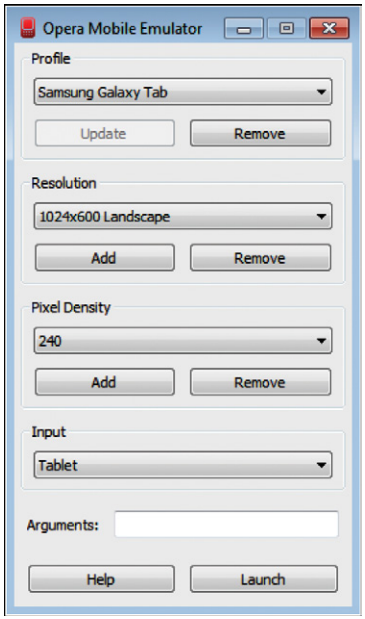


FIGURE 4-11 Selecting an emulator.

With all this said, it is worth making this point even clearer: you should test your mobile site on a variety of mobile devices. In particular, don't trust too much the apparent magic of UA switchers.

Let me illustrate a specific scenario. The UA switcher forces the browser to send a particular UA string and, subsequently, your server-side code recognizes the requesting browser as mobile. This is the only certain fact.

At this point, your server-side code may inject some JavaScript that checks the browser's capabilities and adapts the markup. Your JavaScript code, however, will be querying the desktop browser, not the mobile browser! Furthermore, a generic emulator like the Opera Browser Emulator may not be able to imitate the real browser perfectly, which makes the test experience significantly less valuable.

The bottom line is that you should use switchers for quick testing, then move to device emulators, and finally test on a selection of actual sample devices—the only source of truth.

Note If you want to test your mobile views on specific devices (well, mostly smartphones), all you need to do is get a hold of the same emulators that you use for testing native applications and then use the browsers that they offer.

In addition to buying or renting as many devices as you can, you can use a device-testing service such as Keynote Device Anywhere (*http://www.deviceanywhere.com*). Another tool that is great for remote debugging is Weinre (*http://phonegap.github.com/weinre*).

The Device-Detector Site

As an example of a mobile site that serves device-specific content, let's consider what it takes to build a device-detector website with ASP.NET MVC. Figure 4-12 shows the desktop version of the site that is being made mobile. It has a three-column layout and a variety of links. This site doesn't do any particularly fancy things: it is limited to displaying the UA string and a few other browser capabilities.

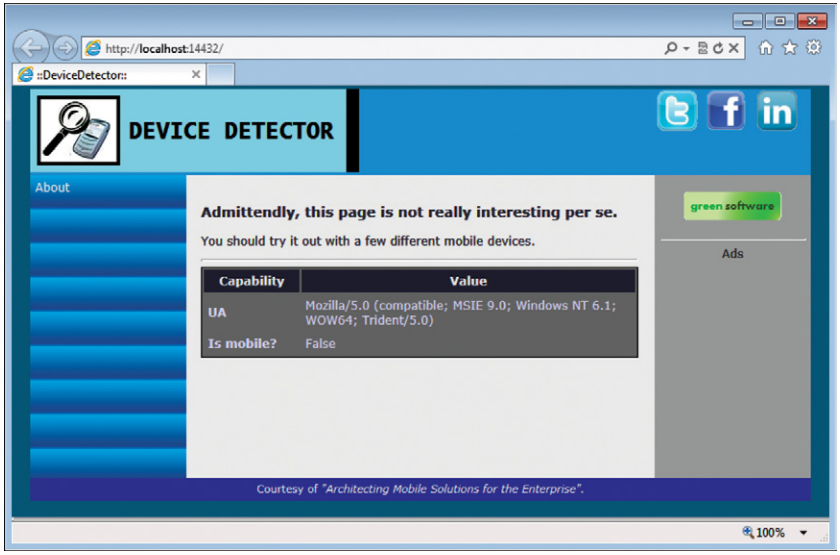


FIGURE 4-12 The desktop version of the Device-Detector sample site.

A mobile version of this site will remove a lot of bells and whistles and mostly focus on the primary use-case: providing details about the current browser.

You should expect to see a single-column template with some graphics at the top (header) and bottom (footer). In addition, a couple of links in the main content area just connect users to the page with device details and contact (see Figure 4-13).

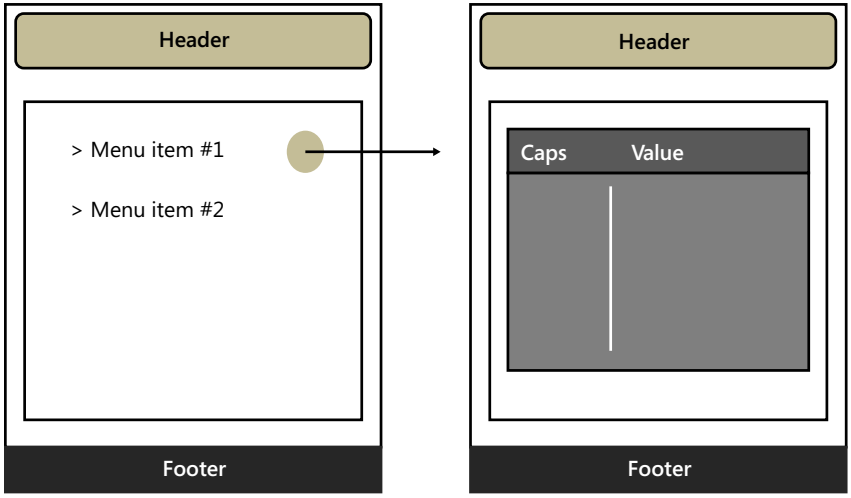


FIGURE 4-13 The mobile layout for the Device-Detector sample site.

In the early days of software development, about 20 years ago, it was natural to design the main screen of applications (mainframe and desktop) through a main menu that accepted user selections via mouse clicks or keystrokes. Nearly all applications were designed around a main menu that collected the various use-cases of the application.

The advent of pop-up and floating menus marked the end of this approach, which clearly was limited to moderately complex applications. With mobile sites (and to some extent, also with native applications), this old-fashioned model is revamped due to the relative simplicity of the logic and the need to be direct and focused.

Routing to Mobile Views

For a site that serves mobile content, an accurate detection of the device is key. In addition, a (possibly) automated mechanism to route requests to the right view would be welcome. Up to ASP.NET MVC 3, you have no tools to allow you to select the view and no convention-over-configuration (CoC) approach.

In ASP.NET MVC 4, instead, an enhanced infrastructure lets you give mobile views conventional names that are resolved automatically by the system. For example, in ASP.NET MVC 4, a view named *index.mobile.cshtml* will be used to serve requests for the *Index* action when coming from a generic mobile device. ASP.NET MVC 4 is expected to give you even more control over the presentation layout, as it also supports syntax like *index.iphone.cshtml* for a specific class of mobile devices.

More Info For more information about MVC 4, refer to *http://www.asp.net/mobile*. In addition, you might want to look at my *Programming ASP.NET MVC* book for a deep coverage of ASP.NET MVC 3.

Configuring a Mobile-Aware View Engine

The source code that comes with this book provides a sample view engine that works well with ASP.NET MVC3—the *AmseViewEngine* class. The engine builds on top of the built-in *Razor* view engine and allows you to invoke both desktop and mobile views. If the browser is detected as a mobile browser, the engine will attempt to resolve any view named *Xxx* as a view named *Xxx.mobile*. If the mobile browser belongs to a specific class, then it is mapped instead to an *Xxx.profile* view, where *profile* indicates the name of the class.

You register the mobile-aware view engine in *global.asax,* as shown in this code:

```
public static void RegisterViewEngines(ViewEngineCollection viewEngines)
{
   viewEngines.Clear();
   viewEngines.Add(new AmseViewEngine(new AspnetMobileViewResolver()));
}

protected void Application_Start()
{
   ...
   RegisterViewEngines(ViewEngines.Engines);
}
```

The constructor of the view engine receives a user-defined class that knows the strategy to transform the originally requested view to see if it exists in the site. In particular, the *AspnetMobileViewResolver* class checks for *xxx.mobile* instead of *xxx* if it detects that the requesting browser is mobile.

View engines are a specific feature of ASP.NET MVC. In ASP.NET Web Forms, you can achieve the same result by registering an HTTP module that intercepts incoming page requests, detects whether the requesting browser is a mobile browser, and redirects to a corresponding mobile page, if any.

Note The *AmseViewEngine* can be extended in a fairly easy manner to look for mobile views in a distinct folder, such as Mobile. All you need to do is change the default value of the *ViewLocationFormats* and *PartialViewLocationFormats* properties to point them to your new Mobile folder. You set the properties in the view engine constructor.

Routing to Mobile Resources

Just as any other type of view, a mobile view may rely on a bunch of external resources, such as images, style sheets, and script files. Because you control the markup of the mobile view, you can simply place all the references you need there. For example, you can link the jQuery Mobile library

only from the mobile layout; likewise, you might want to use smaller images for a mobile view or, better yet, you might want to embed images in the same view as Base64-encoded strings.

It is crucial to note, however, that mobile views may have their own set of resources. Most of the time, you don't need to differentiate mobile resources on a per-device-class basis.

Detecting Device Capabilities

So now there is a mechanism that can redirect automatically to a view that has been specifically designed for mobile browsers. But which part of the ASP.NET run time determines whether the requesting browser is a mobile browser? And, more important, which algorithm is used?

As you may understand, this is the central point of mobile site development—you may not need detailed information in all cases about what a given device can or cannot do, but you always need to know—with extreme accuracy—at least whether the requesting browser is mobile or not.

ASP.NET Native Detection Engine

ASP.NET has its own detection API centered on the following code:

```
HttpContext.Request.Browser.IsMobileDevice
```

The *IsMobileDevice* property returns a Boolean value and indicates whether the current request comes from a mobile device.

Without beating around the bush, this code is not really reliable. For example, it fails on a number of popular devices, such as the HTC Desire and Samsung Galaxy S smartphones (both equipped with Android); it also fails on a wide range of simpler devices, such as the Samsung GT S3370 Corby. Curiously, the native ASP.NET detection API succeeds with the BlackBerry, iPhone and iPod devices, and with Windows Phone devices. Why is this so?

The value returned by the *IsMobileDevice* property results from a partly accurate analysis of the UA string that ASP.NET performs under the hood. Essentially, ASP.NET uses the UA string as a key to match the requesting browser to one of the predefined device profiles. A *device profile* is a text file with a *.browser* extension located on the server under the Windows folder at the following path:

```
// This is the path if you have the .NET Framework 4 installed on the server
\Microsoft.NET\Framework\v4.0.30319\Config\Browsers
```

Figure 4-14 offers a preview of the typical content of this folder.

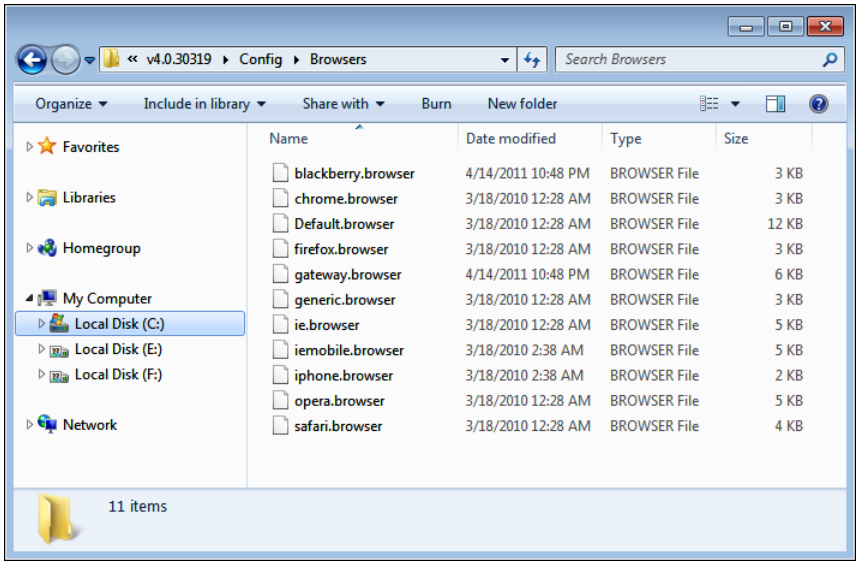


FIGURE 4-14 The content of the Browsers folder.

Files in this folder contain some basic information about a few families of browsers. Each family contains a regular expression used to match the UA string. If a match is found, then the dictionary of browser capabilities exposed to the application is filled with the values of the properties that are known to apply to that family of devices.

This solution was devised years ago when the problem was detecting just a few desktop browsers. Matching the UA string of a mobile device—and its nearly infinite variations—and identifying the right value for a given property require a much richer and articulated database. The content of browser files can be extended and new files can be created, but it doesn't change the basic fact that something stronger is needed.

The bottom line is that if you simply rely on the *IsMobileDevice* property, you seriously risk offering a desktop site to many mobile devices. Worse yet, this especially happens with older devices that will display just a basic site, to the frustration of users. What else can you do?

A Better Way of Detecting Mobile Devices

Detecting device capabilities is a difficult problem in mobile (not just ASP.NET mobile) because of the wide fragmentation of devices. In my opinion, the device fragmentation problem has just one *exact* solution that I'll discuss thoroughly in Chapter 6. This solution is using a DDR like the Wireless Universal Resource File (WURFL).

Most DDRs are not free for every use; and free versions of most DDRs just reduce severely the number of capabilities that they return. The bottom line is that you should be ready to spend money when it comes to DDRs; your money will be well spent.

Anyway, I'd like to illustrate quickly a couple of totally free (but possibly only approximate) solutions that you might want to consider before you go to Chapter 6 and pick up your favorite DDR framework. I'll leave it up to you whether any of these options may work for you in the real-life battlefield.

The first option entails writing your own wrapper around the *Browser.IsMobileDevice* property. You can create it as your own class, or perhaps as an extension method to the *Request* object. Regardless of these implementation details, what really matters is the logic that you use to write the code. Here's a sample detection function written as an extension method for the native ASP.NET *Request* object:

```
public static class RequestExtensions
{
    public static Boolean IsMobileDevice(this HttpRequestBase request)
    {
        var response = request.Browser.IsMobileDevice;
        if (response)
            return true;

        // If response is false, there are still good chances to have a mobile device.
        // Let's check the user-agent string for common substrings.
        var userAgent = request.UserAgent.ToLower();
        response = userAgent.Contains("opera mini") ||
                   userAgent.Contains("mobile") ||
                   userAgent.Contains("samsung") ||
                   userAgent.Contains("nokia") ||
                   userAgent.Contains("htc") ||
                   userAgent.Contains("android") ||
                   userAgent.Contains("windows phone") ||
                   userAgent.Contains("midp") ||
                   userAgent.Contains("cldc");
        return response;
    }
}
```

If the *IsMobileDevice* native property returns *true,* then you can be sure that the requesting browser is really a mobile browser. The problem is with false negatives. The simplest thing you can do is check the UA string looking for common substrings associated with mobile agents. The preceding example includes some manufacturer names and operating system names. The MIDP string refers to the Mobile Information Device Profile (MIDP), a specification that is part of the Java Platform Micro Edition (Java ME) framework. MIDP works on top of the Connected Limited Device Configuration (CLDC), which is instead a lower-level specification. In the end, both MIDP and CLDC are strings that appear often in UA strings sent by mobile devices.

Finally, you may have noticed that this list contains no reference to popular devices such as iPhone (and iPod/iPad) and BlackBerry. This is because ASP.NET 4 comes with *.browser* files for detecting both platforms (see Figure 4-14). As a result, the basic *IsMobileDevice* property works on iOS and BlackBerry devices.

A more powerful approach is based on a repository of mobile profiles that Microsoft built for internal purposes and made public through a CodePlex project: *http://mdbf.codeplex.com*. It's called the Mobile Device Browser File (MDBF). If you visit the website, however, you find out that it is a

dead project now. This means that the database for this project won't be maintained and extended any longer, although you still can download and use it. With the wave of new devices being released every month, this is clearly a problem.

To use the MDBF repository, all you need to do is copy the file in the App_Browsers/Devices folder of your website. The content of the file will be read by the ASP.NET infrastructure and used to populate the dictionary of browser capabilities.

You can extend the MDBF repository—an 18 MB XML file—to keep it up to date. Likewise, you can update *.browser* files and add new ones. Both options, however, are not compelling because they require a lot of maintenance work and research. And this is probably the reason why approximate and exact solutions exist, and exact solutions are not free.

DDR Options

The quintessential DDR is, without a doubt, WURFL, by ScientiaMobile (*http://www.scientiamobile.com*). WURFL was created in 2002—four years before the DDR acronym was first coined on a World Wide Web Consortium (W3C) mailing list. An open-source community-based initiative, WURFL is comprised of an API and a data repository. The API maps incoming HTTP requests to a known device definition and then retrieves known capabilities for that device from the repository. The repository is updated independent of the API so that companies just refresh the repository periodically without the need to change or rebuild the application. The WURFL API is available for a variety of platforms and languages, including ASP.NET, Java, C++, Ruby, and PHP. See *http://wurfl.sourceforge.net*.

In the summer of 2011, the WURFL owners have moved the project to a different licensing model, which is stricter in many ways. While the API is technically still open-source, the Affero GPL v3 license now requires that users completely open-source the proprietary code linked to the WURFl API on their servers. This requirement is typically not compatible with the requirements of commercial entities. Therefore, to avoid open-source provisions, companies can buy a commercial license for WURFL API and data from ScientiaMobile. Note that the WURFL repository is distributed with a proprietary license that prevents you from copying the WURFL data and using it with third-party APIs.

Another interesting DDR specifically aimed at the ASP.NET platform is 51Degrees. See *http://51degrees.codeplex.com*. 51Degrees relied originally on WURFL as the source for device information, but the change in the licensing model of WURFL forced to adopt a different and proprietary repository. Recently, 51Degrees has been relaunched with a new vocabulary (i.e., set of property names) and new data. 51Degrees is a purely commercial initiative, but it offers a free version as a teaser for the platform. The free version is limited to a DDR with just four properties: *isMobile*, *ScreenPixelWidth*, *ScreenPixelHeight,* and *LayoutEngine,* which is simply the browser engine.

Other players in the DDR world are DetectRight (*http://www.detectright.com*) and DeviceAtlas (*http://www.deviceatlas.com*). MobileAware (*http://www.mobileaware.com*) and NetBiscuits (*http://www.netbiscuits.com*) are also names worth mentioning, although they do not offer pluggable DDRs as part of their main business model. In other words, the DDR is just one component of a more elaborate product.

Note Search engines may point you to a few other device detection initiatives, mostly created as an aspect of WURFL and, for this exact reason, subject to legal dispute. Accuracy and level of service, however, don't currently compare to any of the DDRs discussed here.

CSS Media Queries

Lateral thinking is about solving problems using an innovative approach and unconventional reasoning. The lateral thinking about device detection seems to be CSS Media Queries and responsive (or adaptive) web design. Is detecting devices hard? Don't do that, then; just take a bunch of basic properties (e.g., screen size) and let the page adapt and reflow accordingly.

The magic potion that enables responsive web design is CSS Media Queries. Introduced with CSS 3, media queries simplify the design of sites that might be consumed through devices of different screen sizes ranging from 24 inches on a desktop monitor to 3 inches on most smartphones. Media queries are not specifically a technology for mobile development, but the flexibility of this feature makes it really compelling to use to serve different devices with a single codebase.

The idea, in fact, is that you just create one site with a single set of functions and then apply different CSS styles to it by loading a different style sheet for different media. The great improvement brought by CSS 3 is that a medium (such as a screen) now can be restricted to all devices that match given rules. Here's an example of media queries:

```
<link type="text/css"
      rel="stylesheet"
      href="downlevel.css"
      media="only screen and (max-device-width: 320px)">
```

Placed in a HTML page (or view), this markup links the Downlevel.css file only if the page is viewed through a browser with a width of 320 pixels or less. Note that there's no explicit check on the type of browser, whether mobile or desktop: all that matters is the real width of the screen. (Needless to say, with a screen width of 320 pixels, it can only be a mobile phone or handheld device.) The *only* keyword should be added for the sole purpose of hiding the statement from browsers that don't support media queries. These browsers, in fact, don't understand the media type and go right ahead. The full documentation about media queries can be found at *http://www.w3.org/TR/css3-mediaqueries*.

What's the problem with media queries?

It is a common idea these days that by simply adding media queries to a site, you make it ready for mobile clients. CSS media queries help making the page content more mobile-friendly, but they don't affect other critical areas, such as the number of HTTP requests per page, whether DOM manipulation and AJAX are supported, or if a touchscreen is available.

Media queries can check out only a limited number of browser properties—namely, those listed in the W3C standard: device width and height, orientation, aspect ratio, color depth, and resolution to name the most frequently used ones. Most properties support the *min-* and *max-* prefix to help you write more precise queries. Being a CSS feature, media queries only can hide elements that are

too big or low-prioritiy to display on a small screen. You still pay the costs of downloading or keeping these elements in memory. You can use some JavaScript in the pages to download or configure images programmatically. In this way, heavy elements can be managed in a more optimized manner.

In addition, media queries require a browser that supports CSS3. So they work on most smartphones, but not, for example, on Windows Phone 7.0. An all-browser solution for media queries is available through a jQuery plug-in that you can get at *http://www.protofunc.com/scripts/jquery/mediaqueries*. However, there's no guarantee that the mobile browser where you may be using this plug-in can really run jQuery.

Browser Capabilities

Detecting whether the requesting browser runs on a mobile device is only the first step toward delivering an adequate experience to any mobile users. Once you have identified a device correctly as a mobile device, you should try to detect the capabilities of the device. For example, you might want to know the version of the operating system, its real screen width and height, whether it supports AJAX, if it can perform some DOM manipulation, and if CSS is supported. In addition, you might want to optimize the user interface in case the device has a touchscreen or is really a tablet.

Finally, have you ever tried to visit a site with an older phone? If you have, then you know what I mean. First, the phone will likely have no support for WiFI, so it will connect over the mobile network. The slow connection will take a lot of time to see how many different connections are made to download images, scripts, and auxiliary files. You might always want to merge CSS and minify scripts, but what about images? Sprites are a possible solution, but they require CSS support from the device. Inline images (namely, images embedded in the page as Base64 strings) are another route to explore.

You need to know these and possibly more details about the specific device. Some properties can be tested programmatically with a bit of JavaScript. The following code, for example, shows how to check programmatically whether AJAX is supported:

```
var xhr = window.ActiveXObject ? new ActiveXObject("Microsoft.XMLHTTP") : new XMLHttpRequest();
if (xhr === null)
    alert("No support for Ajax");
```

Many other properties can't just be tested programmatically. For instance, how would you detect programmatically if a device understands inline images, has a touchscreen, or is a tablet? For these and other types of capabilities, you need a repository of information that is updated weekly, if not more frequently. If delivering a great user experience on a variety of mobile devices is your goal, then you need the appropriate tool. And you probably need to pay for it.

Putting the Site Up

At the end of the day, a mobile site is just a website that has been designed according to a different set of guidelines. Once you know whether the device is mobile and what its capabilities are, you can proceed safely with the actual design of the site—layout, style, and markup.

Adjusting the Layout

In a mobile site, you might want to use mostly a single-column layout and move navigation and search functions to the top and bottom of the page. Finally, you might want to leave the user free to scroll vertically to locate what is relevant but doesn't fit on the physical page. Beyond these basic rules, the design of a mobile site is all about finding the most friendly and creative way of presenting your content. Here's the layout file for the mobile version of our site.

Note that the listing uses the ASP.NET MVC *Razor* syntax to describe the view. The *Razor* syntax mixes plain HTML with executable expressions. Executable expressions are prefixed with the @ symbol. In particular, in the following example, the *ViewBag* expression refers to a collection through which the page passes values to the view. A good step-by-step tutorial to *Razor* can be found at *http://goo.gl/9eTEm*.

```
<html>
    <head>
        <meta name="viewport"
              content="width=device-width, initial-scale=1.0, maximum-scale=1.0,
                       user-scalable=no" />
        <title>@ViewBag.Title</title>
        <link href="@Url.Content("~/Content/Styles/Site.css", mobile:true)"
              rel="stylesheet"
              type="text/css" />
    </head>
    <body>
        <div id="header">
            <img src="@Url.Content("~/Content/Images/logo.png", mobile:true)"
                 class="image" alt="" />
        </div>
        <div id="content">
            <h2>Know Thy Devices</h2>
            <p>
                Find out details of the devices that visit your site.
                This demo also shows a sample mobile template.
            </p>
        </div>
        <div class="actual-body">
            @RenderBody()
        </div>
        <div id="footer">
            <p>Architecting Mobile Solutions for the Enterprise</p>
        </div>
    </body>
</html>
```

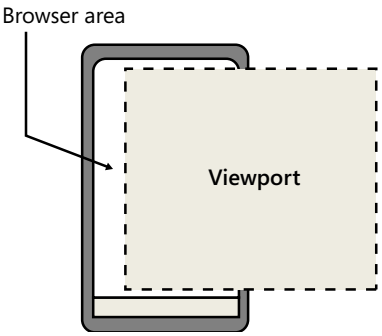
Note that this file is named Layout.mobile.cshtml and is resolved in _Viewstart.cshtml, using an enhanced version of the native *UrlHelper* object in ASP.NET MVC:

```
@using Mobi.Framework.ViewEngines;
@{
    Layout = Url.Content("~/Views/Shared/_Layout.cshtml", mobile:true);
}
```

As mentioned, the layout file implements a single-column view and points to external resources using our custom *Url.Content* method as a resource switcher.

Let's find out more about the *viewport* <*meta*> tag.

Most mobile browsers can be assumed to have a rendering area that's much larger than the physical width of the device. This virtual rendering area is called the *viewport*. The real size of the internal viewport is browser-specific. However, for most smartphones, it is around 900 pixels. Having such a large viewport allows browsers to host nearly any webpage, leaving users free to pan and zoom to view content, as in the following illustration:



This behavior may perhaps be desirable (or at least not too disturbing) when you have a high-resolution smartphone; but what if your users host the site within a 240 × 320 device? It's like looking through a keyhole. To gain control over mobile browsers' viewports, you add an explicit viewport <*meta*> tag and instruct the browser about it as follows:

```
<meta name="viewport"
      content="width=device-width, initial-scale=1.0, maximum-scale=1.0, user-scalable=no" />
```

In this example, you tell the browser to define a viewport that is the same width as the actual device. Furthermore, you specify that the page isn't initially zoomed and can't be zoomed in by users. Setting the width property to the device width is fairly common, but you can indicate an explicit number of pixels. Figure 4-15 shows how the same page looks on an older device with and without the viewport tag.

Adjusting the Style

A mobile site deserves its own CSS file where you define a bunch of new styles required by the specific user interface. In addition, the CSS file will probably need to override some of the styles shared with the desktop site (if any).

For example, you might want to remove background images and replace them with solid colors. Background images are a great example to show how CSS media queries can hardly be the perfect fit when you need to provide multiple views for devices other than smartphones. With media queries, you only have the device width to distinguish devices. However, when it comes to devices less than 300 pixels wide, you can still find different capabilities. Some relatively powerful devices fall in this category that have touchscreen and good HTML capabilities. For these devices (e.g., Samsung Corby),

you can still use background images, but you cannot for any devices smaller than 300 pixels. This is to say that you can do a lot with CSS styles, but not everything. Beyond a threshold, you just need to upgrade to another solution as WURFL. (See Chapter 6 for more information.)

FIGURE 4-15 Effects of the viewport tag.

In a mobile style file, it is important to use the padding property appropriately to ensure that clickable elements (in touch-enabled devices) are large enough to accommodate a relatively inaccurate pointing device like the human finger.

Adjusting the HTML View

The sample site, Device Detector, while not realistic in terms of functionality, is an excellent starting point for understanding the role of browser capabilities. Figure 4-16 compares the desktop and mobile versions of Device Detector.

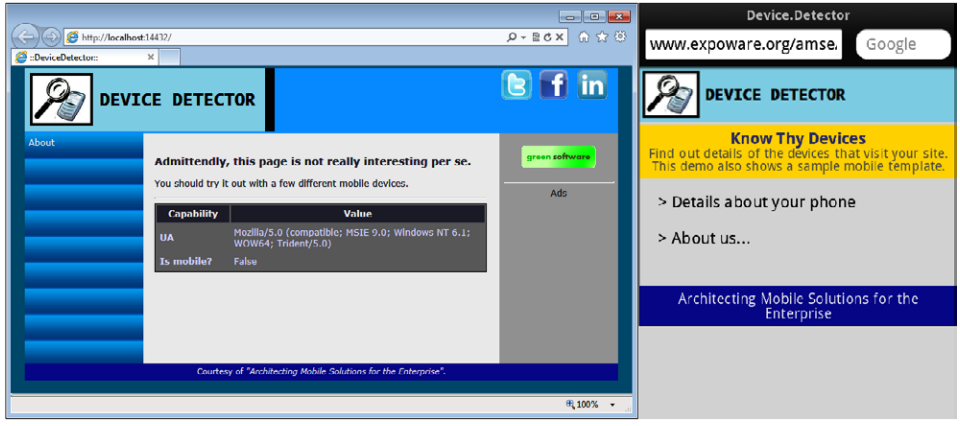


FIGURE 4-16 Desktop and mobile site face to face.

The mobile site requires an extra step to get to the real data: you must click the Details link. The *Index* view is therefore different in our ASP.NET MVC application. The *Index.mobile.cshtml* view simply renders a static markup with a couple of links. It is interesting to see where the *Details* link points. It points to a *Details* action method on a new controller—the *DeviceController*—that is used only by the mobile subsystem:

```
public class DeviceController : Controller
{
    public ActionResult Details()
    {
        ViewBag.UserAgent = Request.UserAgent;
        ViewBag.IsMobile = Request.Browser.IsMobileDevice;
        ViewBag.SupportTables = Request.Browser.Tables;
        ViewBag.MobileDeviceInfo = String.Format("{0}, {1}",
                                                 Request.Browser.MobileDeviceModel,
                                                 Request.Browser.MobileDeviceManufacturer);
        ViewBag.PreferredImageMime = Request.Browser.PreferredImageMime;
        ViewBag.ScreenSize = String.Format("{0} x {1}",
                                           Request.Browser.ScreenPixelsWidth,
                                           Request.Browser.ScreenPixelsHeight);
        ViewBag.SupportAjax = Request.Browser.SupportsXmlHttp;
        ViewBag.DomVersion = Request.Browser.W3CDomVersion;

        // Point to the device.cshtml view
        return View();
    }
}
```

The method would return the view generated from the Details.cshtml template. However, because this method is invoked only from the mobile site, the actual view file will be Details.mobile.cshtml. However, the view engine being used here can pick up Details if it can't find Details.mobile.

The *Details* method collects some data about the requesting browser and composes them into the view. As you can see in this example, the information about browser capabilities is what ASP.NET makes available natively. I actually took the screenshots of this chapter from a site where I installed the MDBF repository—now largely outdated, but far better than the default ASP.NET browser configuration.

In particular, you can consume some of the capabilities being passed down to the view to fork your rendering code, as shown here:

```
@if (ViewBag.SupportTables)
{
    <table id="deviceTableInfo" cellpadding="4" cellspacing="2">
    ...
    </table>
}
else
{
    <ul>
        <li> ... </li>
        ...
    </ul>
}
```

Figure 4-17 shows a screenshot of the site captured from an iPod device.

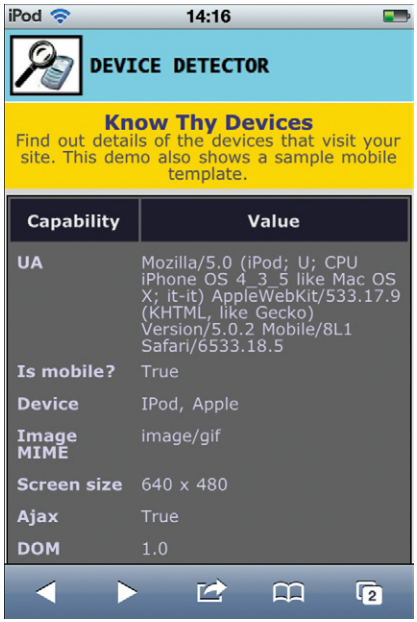


FIGURE 4-17 The Device Detector site displayed on an iPod device.

Figure 4-18, instead, shows the same site on a low-level device, but it is still touch-based and has some decent HTML capabilities (JavaScript and CSS support).

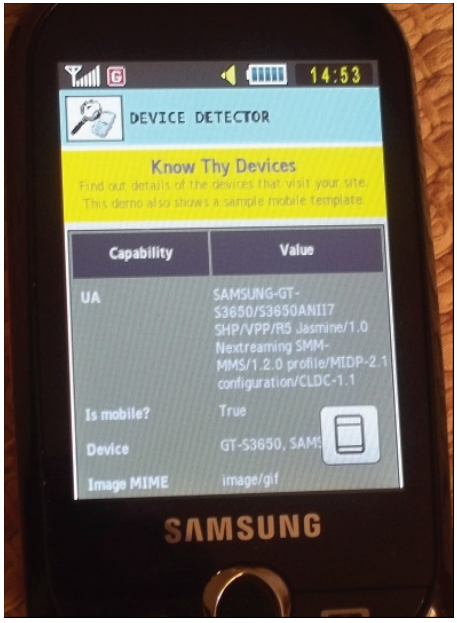


FIGURE 4-18 The Device Detector site displayed on an Samsung Corby device.

Figures 4-17 and 4-18 don't show noticeable differences in the rendered markups. However, if you view it live, you see that the device of Figure 4-18—an older device—takes a while to render the page for at least a couple of reasons. First, it doesn't have much processing power, so any operation (JavaScript or just downloading) is slower. Second, it doesn't have WiFi support; so any download can occur only via 3G. The rendering experience is somewhat painful because it first displays the skeleton of the HTML; next, it downloads the CSS and applies colors; and finally, it gets the picture and inserts that into the layout. Tweaking HTML for older browsers is an operation that you might want to optimize on a per-page basis—if it is worth the cost.

Summary

This chapter tried to turn into practical advice and bits and pieces of code some of the most common practices employed today to build mobile websites. The number of mobile devices is huge, but, on the other hand, you don't have to target all of them. Mobile development is about understanding what you and your customers want and need. This is a crucial point and results in an ideal selection of use-cases. Appropriate and well-described use-cases are essential for any application, but it is a bit more important for mobile applications and sites. For mobile sites, in particular, I estimate that it represents the largest share of the work.

Selection of use-cases helps to keep the whole application up-close-and-personal, establishing a relationship between code and user that is much stricter than with any other type of application.

Beyond this point, building a mobile site is a matter of optimization: minimizing requests, minimizing content being downloaded, and minimizing the user's activity. But it's also a matter of optimizing the design to find a good compromise between UI widgets and touch capabilities. The user interface is critical: common widgets like drop-down lists and check boxes, while valid to express the behavior of a view, may require a different rendering and graphical structure.

Finally, a mobile site may largely reuse code in the back end of the sister desktop site. Coding the back end is probably the simplest aspect of mobile site development. The next chapter covers in more detail two technologies that have been mentioned only briefly up to now: jQuery Mobile and HTML5.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

About the Author

A longtime trainer and top-notch architect, Dino Esposito is the author of many popular books for Microsoft Press that have helped the professional growth of thousands of .NET developers. His latest books are *Programming ASP.NET 4* and *Programming ASP.NET MVC3,* which have been translated into a variety of languages. Every month, at least five different magazines and websites throughout the world publish Dino's articles, which cover topics ranging from web development to software design practices, and from mobile development to ASP.NET Model-View-Controller (MVC) and social network development.

An ASP.NET Most Valuable Professional (MVP), Dino is available for onsite consulting and training on web and mobile development and software practices. When traveling, Dino is often the guest star of user-group meetings in Europe. If you run a user group, feel free to get in touch.

In the rest of his everyday working life, Dino is the CTO of Crionet (*http://www.crionet.com*), a fast-growing company providing software and mobile services to professional sports, especially tennis. Dino led a team that created a range of mobile apps for Android, iOS, Windows Phone, and BlackBerry, such as the official app for the Rome ATP Masters 1000 tournament. Dino also contributed to the popular (and multiplatform) Postino app (*http://www.postinoapp.com*) for sharing real postcards from mobile pictures, and writes for the Mopapp technical blog (*http://www.mopapp.com*).

Dino speaks regularly at industry conferences all over the world, including Microsoft TechEd, DevConnections, and premiere European events such as DevWeek, Software Architect, and BASTA. He is fairly active on social media; you can follow Dino on Twitter as @despos, and read his mobile blog at *http://www.mopapp.com/blog*. The blog focuses on a wide range of mobile-related topics, including native app planning and development, sales monitoring, patterns and strategies for the various platforms, mobile site development, responsive Web design, smart TV programming, HTML5, and appstore interactions.

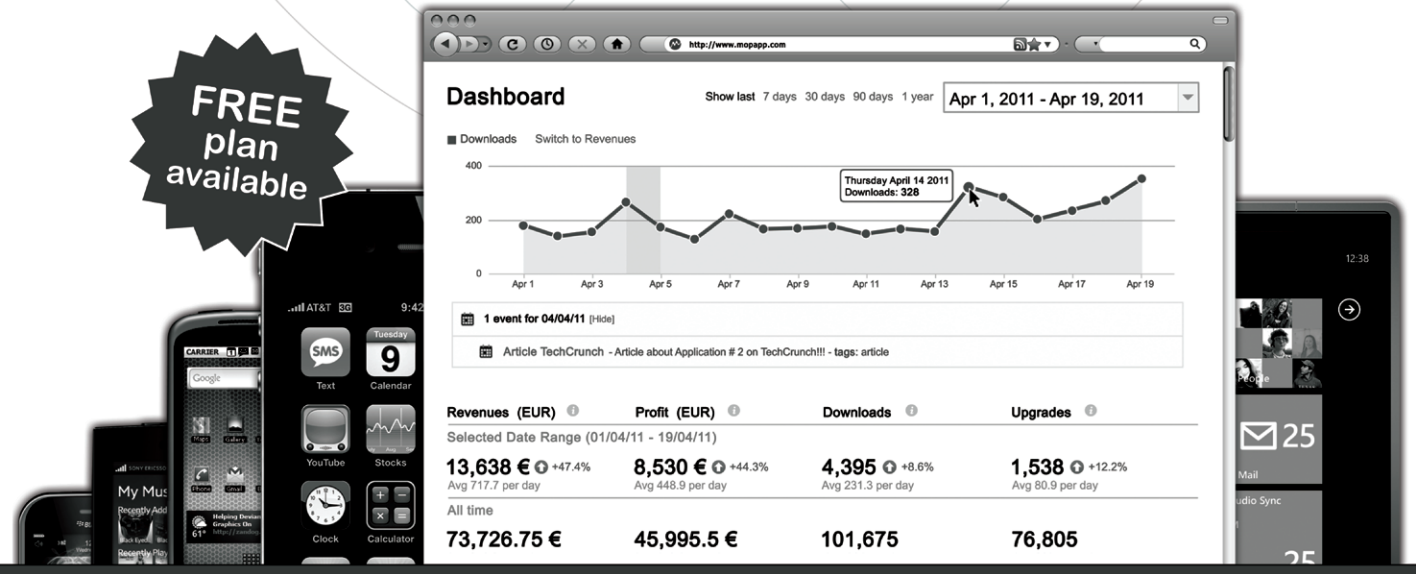
Finally, Dino makes every reasonable effort to become a better domain expert in tennis. This means watching tennis live and on TV, and planning new applications—but especially playing tennis on dusty clay courts at CT Monterotondo, in Monterotondo, Italy.

mopapp

The only enterprise-level solution for mobile apps analytics

- import & export sales and data through Mopapp API
- licensing option: match your company's look & feel and use your own sub-domain
- tailor-made import of legacy data
- track unlimited apps and in-app items
- automatic integration with all major app stores
  - Handango
  - Google Play
  - WP7 Marketplace
  - RIM App World
  - Amazon Appstore
  - iTunes App Store
  - Samsung Apps
  - MobiHand
  - GetJar
  - Barnes & Noble
- lots of reports with advanced filtering and grouping
- monitor user reviews translated to your own language
- export to Excel and PDF
- subaccounts & report sharing
- automatic integration with all major ad networks

FREE plan available



What do you think of this book?

We want to hear from you!

To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Tell us how well this book meets your needs—what works effectively, and what we can do better. Your feedback will help us continually improve our books and learning resources for you.

Thank you in advance for your input!

Microsoft® Press