The Mobile Web Handbook

Published 2014 by Smashing Magazine GmbH, Freiburg, Germany.
Printed in the EU. ISBN: 978-3-94454093-1.

Cover Design, Illustrations and Layout by Stephen Hay.
Copyeditor and Proofreader: Owen Gregory.
Editing and Quality Control: Vitaly Friedman.
eBook Production: Cosima Mielke.
Typesetting: Markus Seyfferth.

The Mobile Web Handbook was written by Peter-Paul Koch
and reviewed by Stephanie and Bryan Rieger and Vasilis van Gemert.

Links and updates of this book can be found at
http://quirksmode.org/mobilewebhandbook.

Table of Contents

Introduction

Introduction

The Mobile Web Handbook explores the differences between mobile and desktop web development that we should be aware of when creating websites for both. It's not very technical — there are only a dozen or so simple code examples. It discusses no libraries or tools. It's about mobile web fundamentals.

There is no mobile web distinct from the desktop web. Developing websites for mobile is pretty much the same as developing for desktop, especially now that responsive design techniques allow us to adapt our CSS layouts to both huge desktop screens and tiny mobile ones.

Still, there's "The Mobile Web" in the title of this book, and that's not an oversight or marketing trick. It serves as a convenient shorthand for "touch-based small-screen web on more browsers than you've ever heard of." Mobile web development is not fundamentally different from desktop, but there are subtle distinctions that may cause you trouble if you're unaware of them.

It's best to see mobile web development as a layer that you apply on top of regular web development, and which contains a few new concepts and techniques that you must understand in order to create compelling mobile experiences. This book concentrates on that mobile layer, and highlights three topics:

1. On desktop we have only five browsers, but on mobile it's more like 20 or 30. These are not all separate browsers: many of them are subtly different versions of the same browser, especially of Android WebKit. Why is that? How do you handle it? Why is Android so complicated? How will the mobile browser market develop?

2. On desktop, there's only one single viewport: the browser window. On mobile, this viewport was split into two, and a third viewport was added. Why do we need three viewports? How do they work?

3. Desktop has its keyboard and mouse events, and touchscreen browsers need special JavaScript events to react to the user's touch actions. This may seem logical but Microsoft, of all companies, challenged that logic and raised interesting philosophical questions about the relationship between JavaScript events and interaction modes. On a practical level, the touch events have some special features that you need to know about.

Browsers, viewports, and touch events are the main themes of this book. There are also a few smaller items: the rise and fall of browsers and operating systems; what proxy browsers are; why a few CSS declarations such as `position: fixed` are more difficult to get right on mobile than on desktop; and becoming an accomplished mobile web

developer by setting up a device lab and reconsidering outdated development practices. As a bonus, you will learn *why* responsive design works. (Not *how*. You already know how. But do you know *why*?)

So here we go. It's going to be quite a journey.

What This Book Doesn't Cover

In order to manage your expectations, here are a few topics that are not treated in this book. This is about the mobile web, so there is no information on native apps. You can use this book for creating hybrid apps (that is, apps written in HTML, CSS, and JavaScript but wrapped in native code), but only for the web component, not for the native one.

I'm not a designer, so I don't say anything about design except for some very vague general tips. No design patterns, either.

The mobile market is very volatile, and browsers and devices that are a hit now could be a memory in a year's time. That's why I try to steer clear of inspecting individual devices and browsers, though sometimes I make an exception for Safari on iOS because it's so very influential on web development thinking.

Finally, the most complicated caveat: this book only investigates fundamental differences between desktop and mobile, and generally ignores topics such as AppCache, which, though more important on mobile than on desktop, are not unique to mobile. This is sometimes a subtle distinction, but it helped me a lot in keeping the scope of this book, and of my research, to manageable levels.

Companion Site

Writing a book about the mobile web is challenging because it's one of the fastest-changing environments ever — faster by far than the traditional desktop web. I write this in summer 2014, and by the time you read it things will have changed. That's why I try to concentrate on fundamental issues and problems, and don't pay too much attention to quick-shifting details such as browser bugs.

Still, you need to know about the bugs as well. That's why I created a companion site at http://quirksmode.org/mobilewebhandbook that contains links to my browser research to back up what's in this book — or, as time progresses, to show which mobile browsers have mended the errors of their ways, or changed, or done something else of note.

In this book I occasionally give browser compatibility notes, but more often I'm rather vague; for instance, saying that "many browsers" support this or that. The companion site always gives a breakdown of those browsers, and includes notes on bugs.

Tablets

The Mobile Web Handbook focuses on mobile devices; that is, small devices that fit in the palm of your hand and can establish a connection over a mobile network. It does not really cover tablets or other types of devices.

Still, a lot that's in the Handbook also applies to tablets. Tablets, too, have touch-based browsers, and although they have larger screens than mobile phones, they're still smaller than most desktop screens and have three viewports instead of one.

Besides, what exactly is a tablet? Samsung, in particular, tends to bring out more and more very large phones, which you can easily see as small tablets instead. The Microsoft Surface is a tablet with an attachable keyboard, which converts it more or less into a laptop computer.

Is the Samsung Galaxy Note 8.0, released in Q2 2014, a huge phone or a mini tablet? Or is the distinction meaningless?

Right now we can't tell if tablets are going to remain a separate device category, or if they'll quietly fold into the phone and laptop categories. From a technical perspective it doesn't really matter, though. Tablet browsers are mobile browsers in all respects, and obey the same rules and restrictions. Although this book will hardly mention tablets again, you can safely assume that anything you build for mobile will work on a tablet as well, with the obvious caveat that a tablet screen is bigger than a phone screen and your responsive design should accommodate that.

Thank Yous

This book didn't spring from my forehead fully formed. Plenty of people were involved, and I'd like to thank all of them. Vitaly Friedman saw the potential of this book, signed me up, and was the general editor for all chapters. Markus Seyfferth arranged all practical matters such as contracts and printing. Stephanie Rieger was good enough to be the technical editor for all chapters. Stephen Hay signed on for the cover, illustrations, and overall book design. Patrick Lauke edited the Touch and Pointer Events chapter, a topic he knows more about than most other web developers I know combined. Max Firtman went over the Browsers and Android chapters and provided valuable feedback.

Then a compelling presentation by Jason Grigsby and a discussion with the MSIE team caused me to overhaul the Touch and Pointer Events chapter once more. Finally, Vasilis van Gemert read through the entire second draft from the perspective of a teacher, while Owen Gregory signed up for those last finicky copy edits that make a good book a great one. Thank you all, ladies and gentlemen. The book wouldn't have been as good as it is now without your timely help. All remaining errors are, unfortunately, my own.

***

Now let's get started with a general overview of the mobile world. You'll find that it differs a lot from the desktop world we're used to.

Chapter 1

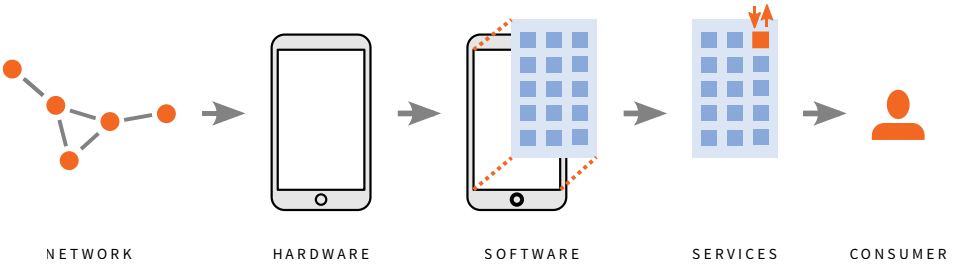
The Mobile World

Chapter 1

The Mobile World

In order to understand mobile web development we have to understand the mobile world. Where the desktop situation is pretty well understood, mobile is so different that it pays to examine it in detail and carefully note how it's different. Not only will that explain why certain browsers are more important than others, it will also make you sensitive to several issues that don't play a role on desktop at all but are vital on mobile. In particular, the role of the mobile network operators is quite different from the desktop ISPs.

The Mobile Value Chain



The mobile value chain extends from the network operators, via device vendors, software makers, and service providers to the consumer. This chapter will study the first three links in the chain.

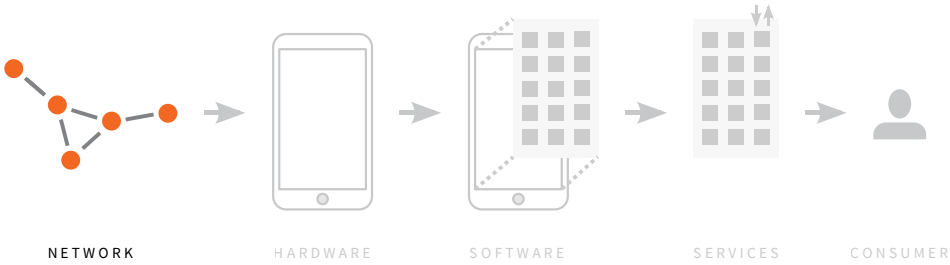
Traditionally, the mobile value chain was formed by operators (called *carriers* in the US) and device vendors. Recently, operating system vendors entered the value chain, and they are being followed by service providers. In fact, the software and service layers are rapidly gaining in importance, and thus software vendors such as Google and Microsoft, and service providers such as Facebook and WhatsApp are becoming equal partners to operators and device vendors.

Each link in the chain enhances the value of the others. For example, a mobile network is worthless without mobile phones, and vice versa, while mobile phones can't do without a first-rate operating system and important service apps. Thus the four parts of the mobile value chain are dependent on one another. Despite that, they sometimes act against one another because each of them has the same goal: commanding consumer mindshare and money. Each of them would love to lock the consumer into a vertical silo of its own making, where anything the consumer does is controlled by the company and makes money for the company. (Apple is, of course, the most successful example.)

At the same time, operators and device vendors fear becoming commoditized; that is, becoming indistinguishable from their direct competitors. If all network connections are the same, why would consumers care which operator they're with? If nearly every phone runs Android, why would consumers care what kind of phone they buy? They don't, and that's why the Android vendors have their minds set on differentiation. We'll get back to that.

On a large scale, studying the mobile market is mainly a process of predicting which companies — and which parts of the value chain — will be more successful than others in avoiding commoditization.

Operators



The operators own and maintain the mobile networks. Until now, they were the winners of the mobile game because they made incredible amounts of money, especially on text messages, and they dominate the consumer market by subsidizing devices.

Operators have no points of differentiation: in the end, consumers care very little whether they're on Vodafone's or T-Mobile's network. Besides, operator profits are falling as a result of changing consumer habits: customers are sending fewer text messages, preferring to use other IM solutions such as BlackBerry Ping and WhatsApp.

Some operators understand they have to work with (web) developers and offer them APIs for payments, like Blue Via does; but not all do so, and even the ones who do have to compete with Apple's App Store and

Telefónica's Blue Via initiative (http://smashed.by/bluevia) offers a fairly easy API for mobile payments. Purchases from developers are paid via the monthly invoices Telefónica sends to its clients anyway. Better, clients don't have to register because they're already registered with Telefónica. The disadvantage is that it only works on Telefónica's networks and a few others. Operators could play a major role in online payments, but so far haven't built a global payment system — and time is running out.

Google Play, which have become the standard for mobile purchasing. The operators, then, are in trouble. I expect them to gradually become less important, as other mobile players, especially device, OS, and service vendors, win consumer mindshare — and the consumer money flow. At the time of writing, though, they still have a powerful position.

Connection Providers

In a large part of the world, operators are just connection providers. However, in many, but not all, developed countries, they have a much larger role, actively deciding which devices consumers will get.

Let's start with the simpler case. In most developing nations, average consumers buy a phone at a specialist store, or a Samsung, Nokia, or other branded store and, once they have the hardware, get a SIM card, usually pre-paid, at another store. They top up their SIM card when it's necessary (and they have the money). In fact, many consumers get more than one SIM card. They search for the best deal for voice, then for SMS, and possibly in future for data as well, and switch networks based on what they want to do right now. This, in turn, makes dual (or even triple) SIM card devices popular: consumers aren't going to manually switch SIM cards several times per day.

In this model, operators are not all that powerful. They offer a service and compete against other companies offering the same service, with consumers paying avid attention to pricing and quality. Simple.

Operator Subsidies

In many developed countries operators play a quite different role, since they actually sell phones to consumers and offer a subsidy.

These subsidies are a powerful weapon because the psychological mechanism behind them is so devious that nearly all consumers fall for it.

Operators offer phones to consumers for a lower price than they can find elsewhere. If you buy a new high-end Android phone in the operator's store, you might pay only €100 or so, while the normal sales price is more like €600. Of course, the operators don't give you €500 out of their own pocket: they earn it back (with interest) on the two-year contract that you're obliged to sign.

Although operator subsidies exist in many developed countries, sometimes they're forbidden by law, for instance in Belgium and Italy. Here consumers buy phones and subscriptions separately.

Although buying a smartphone for the full price and a separate contract for connectivity is cheaper in the long run, the psychological difference between €100 and €600 is so huge that most consumers don't even think about buying a phone anywhere else than in an operator's store. (My sister saw through the operators' cunning plan without my having to brief her, and bought her iPhone directly from Apple. I was very proud of her. But she's an exception.)

Here's something you should do every few months. Go to an operator's store, pretend you know nothing about smartphones, and ask for advice on purchasing a phone. The store clerk will efficiently steer you towards the type of phone that the operator currently wants you to buy.

Store clerks earn a slight commission on any phone they sell, but the exact amount depends on the type of phone. By changing the commission, operators make sure clerks drive consumers toward the devices they want to be sold right now.

At the time of writing, that device is always an Android phone, and often a Samsung. Consumers are familiar with the brand, and most Android vendors are able to produce phones fairly cheaply due to economies of scale. This lower price frees up extra money for the bottom line — and even a little for store clerks, independent resellers, and consumers.

Through this process, operators gain considerable power over device vendors. If the operators decide that they don't want to sell certain types of phones, they can simply remove them from their stores. Sometimes they're contractually obliged to offer the phones, but in that case they place the devices at the back of their stores and slash the clerks' commission, with the obvious result that nobody buys them anymore.

The takeaway for us web developers is that by deciding which phones will be offered to unsuspecting consumers, operators influence the mobile browser market, because those devices' default browsers will get more market share. Thus, keeping track of operators' current preferences is important.

Subsidies or No Subsidies?

So what's the difference between providing a subsidy and not providing one? Obviously, subsidies and the commission system give operators more power over device vendors, which usually translates into lower device prices for them (and they get bulk discounts anyway).

Also, the subsidies cause more high-end devices to be sold, since more people (think they) can afford them.

Conversely, many consumers in unsubsidized countries opt for mid-range or cheap phones, because there is no subsidy and people in developing countries have less disposable income. Since consumers have to pick a phone themselves, brand awareness becomes more important.

When confronted with devices of a similar price, will the average consumer pick a Samsung or a Nokia? Device vendors try to influence consumers' brand awareness through ad campaigns and flagship stores. Samsung, in particular, has the advantage that its brand is also known in related electronics fields like TVs and household appliances.

Nokia and Samsung have specific phones for both types of markets. Vendors that exclusively create expensive high-end smartphones, such as BlackBerry and HTC, have more trouble in the unsubsidized markets, although BlackBerry is still fashionable in some countries, such as Indonesia. In general, though, the subsidized markets are more important for these vendors, which makes them more vulnerable to operator whims.

From the operators' perspective, subsidized markets are ideal, since they give them lots of power. As we saw earlier, their big worry is that, like in unsubsidized markets, they will become dumb pipes, only good for transferring data packages from A to B.

Avoiding this fate is their prime purpose right now, and they generally understand that they have to offer something to developers. Unfortunately they're not very good at developer relations, because what they

offer is complicated, restricted to their own network, and may be gone after one or two years, when the next reorganization brings in new managers who want to do things differently simply because they can.

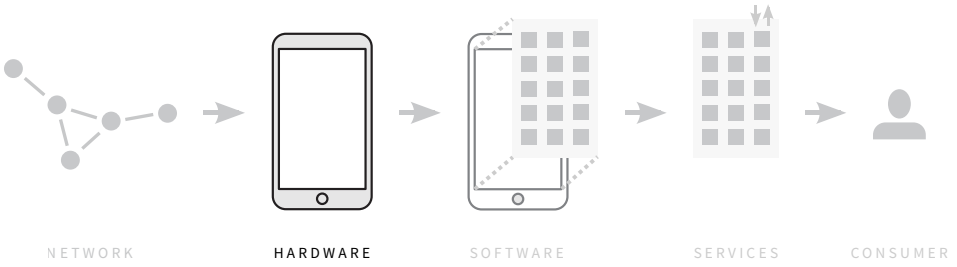
Developer and Consumer Mindshare

And what about Apple? It is a special case, which is why operators don't like the Cupertino giant. As one would expect, Apple's hefty pricing leads to a lower sales market share in unsubsidized countries, but even there sales are decent and growing. Apple is expensive, but it has such a huge brand awareness and customer loyalty that the price tag doesn't really matter. iPhones are becoming status symbols for the up-and-coming middle class in developing countries.

Apple's real power lies in the subsidized markets, though. There, it can break the operators' power over the consumer quite easily, because a small but dedicated (and affluent) slice of smartphone buyers wants an iPhone, and isn't interested in anything else. When a store clerk is confronted with such a consumer he'll give in and sell an iPhone instead of the current operator offering, because a sale is better than no sale. But operators don't like it.

The reason Apple — and only Apple — has this power is because it is popular with both consumers and developers. Google has a lot of developer mindshare, but not so much consumer mindshare. The traditional mobile companies such as Samsung and Nokia have a lot of consumer mindshare, but not very much developer mindshare. Only Apple has both, and that gives it enough power to occasionally ignore operators.

Device Vendors And Hardware



Mobile networks are worthless without mobile phones. Duh. It's time to turn to device vendors and their role in the mobile market.

Device vendors create the mobile hardware, and sometimes the software. Most of them try to cover the entire mobile market, from cheap to expensive, by creating several lines of phones. Obviously, the cheap phones have less powerful hardware and less functionality than the expensive ones.

Following a Phone

Before delving into the details, we need to understand the big picture. So, let's follow a phone from its inception until it ends up in the consumer's pocket. Suppose Samsung decides to produce a new high-end smartphone. The first item on the agenda is to figure out what kind of components it can afford for a reasonably priced phone while still making a profit. Tightly tied in with this is the selection of an OS. The most obvious choice at the time of writing is Android.

Then the phone gets designed: the hardware, the UX, and the changes to the default Android software. Also, Samsung decides which of its own apps it will include as firmware.

Around this time the phone is announced. Marketing copy is created and disseminated through the usual PR channels. Samsung hopes the pending release triggers a wave of interest.

Now Samsung starts negotiating with the operators that are going to subsidize the phone. Bulk discounts, placement in the operator stores, and marketing are discussed. Marketing is even more important in unsubsidized countries, so plans are drawn up.

Then comes the actual production process in Samsung's factories, with prototypes and final versions. Test units are sent to strategic partners, and after a final feedback round the phone is released. Samsung stores, the operators, and the independent stores now get their phones. Usually, this doesn't happen in all countries simultaneously, which is the reason why some phones are not available throughout the world.

If Samsung sells units to operators, there's an extra step: the operators will want to put their own apps on the phone, and maybe customize the start screen, home screen, or even the browser. (One operator once went as far as setting the HTML `<li>` bullets to a color that was not even their brand color — and made sure web developers could not override it. One wonders what they were thinking.)

Now the main marketing campaign starts up. Samsung is dependent on these in the unsubsidized countries. Although in subsidized countries the operators will make sure their customers pick the new phones in their stores, some extra marketing never hurt anyone.

From inception to entrance in the market, the process takes at least six months, and possibly as long as twelve. In general, the larger the com-

pany is, the more bureaucracy, and the more people have to sign off on the phone, which may cause delays. This gives an edge to smaller device vendors, especially Chinese ones such as Xiaomi; because they have faster release cycles they can react more quickly to new trends in the market.

These six to twelve months assume an existing OS: if the OS is new and untested, the process would take about six to twelve more months because several software iterations are needed to get the OS right. (That's why you shouldn't believe any news item that states a phone with a new OS will be released within the next year. It won't.)

What is a Smartphone?

Often, major new releases from Samsung and other long-established device vendors are labeled "smartphones." But what exactly is a smartphone? Why should we care?

The border between smartphone and non-smartphone is somewhat arbitrary; it exists mostly due to historical accident and is unimportant to web developers. We care whether a phone has a browser. Unfortunately, market analysts care only whether it's "smart."

Up until about 2010 mobile phones were divided into basic phones, feature phones, and smartphones. Basic phones can only do voice calls and SMS. Smartphones were unofficially defined as phones that allowed the user to install apps and that ran a recognizible OS. Feature phones were everything in between: phones that offered more features than basic phones, but fewer than smartphones. Specifically, it was not possible for the user to install apps on a feature phone.

Today, the line between smartphones and feature phones is blurred, to the point where almost every phone that can do more than voice and SMS is a smartphone. Unfortunately, we're still using the definitions from 2010. Androids are smartphones because they were classified as such in 2010, while S40 phones are feature phones for the same reason, even though S40 phones allow the installation of apps nowadays and compete with cheap Android devices.

When it comes to web development the line between smartphones and feature phones is entirely arbitrary. Nokia S40 devices have a browser and are used on a massive scale, especially in Africa, where consumers don't have any other internet-capable device. That makes it important to web developers targeting these regions, despite being a feature phone — and despite not being counted at all in smartphone statistics.

This problem will solve itself. Smartphones become cheaper and cheaper, and eventually they'll displace the entire feature phone category. By then anything that can do more than just voice and SMS will be a smartphone. For now, the arbitrary distinction between feature phone and smartphone is one more problem that crops up when discussing mobile market shares.

The Global Device Market

That brings us to the complex question of mobile market shares. What kinds of devices are being sold, and how much of each type? How does that affect mobile web development? I'll provide some numbers, and caveats, later on, but it turns out that these questions are surprisingly hard to answer. It helps if we first discuss the global device market qualitatively.

First, a "duh" moment. In general, the more expensive phones are sold in richer countries, and the cheaper ones in poorer countries. We've already encountered the reasons: in wealthier countries consumers have more money and operators offer subsidies, making it easier to buy a €600 phone; in poorer countries, people have less money and operators don't give subsidies.

Still, this is a generalization. Rich elites in poor countries have plenty of money, so they can afford anything. Up-and-coming middle class consumers might also buy expensive smartphones for status reasons (sometimes in addition to the simpler phone they actually use in daily life). Middle-class people from rich countries might not feel the need for a smartphone and instead buy something cheaper.

Different countries may have different popular brands. While most of the world watched in awe at Nokia's decline and fall in 2010–2013, the US remained cheerily indifferent because Nokia never made inroads in the US market and wasn't an established brand. Motorola is the exact opposite: it's still a power in the US, but irrelevant elsewhere. Despite crashing globally, BlackBerry still has a following in Indonesia, and retains about a 10% share of the British mobile browser market.

General rules don't help us much further. The fundamental lesson is that the so-called *global device market* doesn't exist. Instead, there are dozens of regional markets, and although you can aggregate the data to create global statistics, they don't tell you anything useful about particular markets. There are too many differences in demographics, culture, brand awareness, and disposable income to define general worldwide rules for the mobile phone market.

Finding the Right Stats

One of the challenges of mobile statistic gathering is knowing what kind of data is useful. *Sales market shares* are generally widely available because these numbers are important for investors and the stock market. So we know roughly how many phones are sold annually, and what the vendors' market shares are.

Still, pure sales statistics are not all that important to us web developers. More important is the *installed base share* — what kind of phones people have in their pockets. Where sales stats tell us which kinds of phones consumers will buy in the near future, installed base stats tell us what kind of phones they'll fire up right now if they want to browse.

It's all well and good to know that hardly any Android 2 devices are sold any more in developed countries, but plenty of people still have an old Android in their pockets. Sure, they'll switch to Android 4 when they buy a new phone, but that hasn't actually happened yet. If they want to surf now, they'll use Android WebKit 2 — not the best of browsers. Is your website ready for that?

The installed base of a phone or OS rises more slowly than its sales market share. In the developed world, consumers generally buy a new phone every two years, since that's the length of the average operator contract. Therefore, in any given year at least half of consumers won't buy a new phone but continue to use whatever they have. So if Android takes 70% of yearly sales, at the end of that year only 35% of the consumers will actually have a new Android phone in their pockets; 35% more will likely switch to Android in the following year, but are using something else right now. It's this effect that installed base measures.

But even installed base isn't the real piece of data we want. As we'll see in a moment, Android's sales share in 2013 was 78%, while its installed base was about 65%. Still, its *browsing market share* was only about 35%. The reasons for this discrepancy are hotly debated. Do Android users genuinely browse less than iOS users? Are the sales numbers wrong? Is there an error in the detection scripts?

Although this discussion is interesting in and of itself, in the end we web developers don't care about the entire phone market. We only need to satisfy those people who actually use their devices for browsing. Thus, the Android default browsers (Android WebKit and Chrome combined — we'll get back to this in the Android chapter) merit roughly as much attention as iOS's Safari, which is at about 25%. The fact that Android's sales and installed base share are much higher than iOS's is irrelevant.

Even general reports of browser market shares are not all that important. In the end, what matters is which browsers people use to visit the websites of your clients. So ask your clients for their access logs and study the browser make-up. Aggregate statistics for an entire country should be used only if your client doesn't have any.

Who gathers the global statistics we're about to see? The mobile phone market is analyzed by several companies, but what I use below is the aggregate data created by Tomi Ahonen, a mobile analyst and former Nokia executive. He has a reputation for being a mobile stats hound, and it's very hard to find better statistics that don't come directly from one of the analyst houses. Also, Tomi doesn't champion one OS or vendor over another, which means his numbers are as honest as he can make them. These particular numbers come from http://smashed.by/mwhb1 . The local statistics for OS sales come from a different source indicated with the table.

Smartphone Sales Market Share

Phew. Those were the generic caveats, so we can finally look at some numbers. Here are the market shares of the top 10 smartphone manufacturers in the world for 2013 compared with 2012:

Vendor	Country	OS	2013	2012
Samsung	Korea	Mostly Android	32%	31%
Apple	US	iOS	16%	20%
Huawei	China	Android	5%	5%
LG	Korea	Mostly Android	5%	4%
Lenovo	China	Android	5%	4%
ZTE	China	Android	4%	4%
Sony	Japan	Android	4%	5%
Coolpad	China	Android	4%	*
Nokia	Finland	Windows Phone	3%	5%
HTC	Taiwan	Mostly Android	3%	5%
Others		Android, BlackBerry	19%	17%
Phones sold (in millions)			990	697

Market shares of top 10 smartphone vendors in 2012 and 2013

These numbers are less precise than you'd think. At the time of writing, Samsung is not divulging its exact sales figures — just the total number of all phones it has sold. It's up to analysts to split this number into smartphones (which are counted) and feature phones (which aren't).

Speaking of not counting feature phones, Nokia's S40 platform is entirely absent in these numbers, even though the OS is relevant to web developers because you can browse with it. So Nokia's share of sold web-enabled phones is much larger than the table shows.

Also, you will likely see a few vendors that aren't active in your country at all. The Chinese vendors, in particular, sell most of their phones in their domestic market and in developing countries, and are absent from developed countries. That's what I mean when I say there is no global device market.

OS Sales Market Share

Still, we do get an impression of how many devices with a specific operating system have been sold in 2013. It's not a very exciting metric — as you'd expect, Android easily takes the lead.

OS	Creator	Device vendors	2013	2012
Android	Google	All but Apple, Nokia, and BlackBerry	78%	65%
iOS	Apple	Apple	16%	20%
Windows Phone	Microsoft	Mostly Nokia	3%	2%
BlackBerry	BlackBerry	BlackBerry	2%	5%
Others			1%	7%

Sales market shares of mobile smartphone OSs in 2012 and 2013

The main question here is whether Android has reached its highest OS market share, or if it will grow even beyond 78%. In other words, will the other OSs shrink even more? When forced to guess I'd say Android can grow a little more. Its growth is fueled mostly by cheap Android phones (which are counted as smartphones) replacing older feature phones (which are not counted as smartphones). So the combined market share of the other three will shrink a bit more, I think. Then again, I could be wrong, and even if I'm right my prediction is pretty vague.

Again, these are global statistics. They don't necessarily say anything about your country. To illustrate that, here are the OS sales market shares for four countries and one continent for Q3 2013. (And why did the creators of the table decide to compare a continent to four countries? I don't know.)

These numbers come from WP Central, a Windows Phone-oriented site http://smashed.by/mwhb2

OS	Italy	France	UK	US	Latin America
Android	72%	68%	58%	57%	73%
iOS	10%	15%	27%	36%	7%
Windows Phone	14%	11%	11%	5%	6%
BlackBerry	2%	5%	3%	1%	5%
Others	2%	1%	1%	1%	9%

OS sales market shares in four countries and one continent

Can you spot the differences? Android's share fluctuates between 57% and 73%. Windows Phone has a more-than-decent market share in this set of statistics: from 5% to 14%. That's rather higher than we'd expect from the global numbers and the Windows-Phone-will-never-amount-to-anything story. But is that a data error or are our expectations wrong?

These stats were published by a Windows Phone-centric website. That does not mean the data are false, but it could be that the analyst and the countries were cherry-picked to show Windows Phone successes instead of failures. This doesn't even need to be a conscious decision: if you're a Windows Phone fan you're more likely to republish great Windows Phone numbers than lousy ones.

Then again, it may be that these numbers give a good indication of where Windows Phone is going, and our expectation that Microsoft's OS will never amount to anything is wrong. If a source doesn't conform to your preconceptions, always wonder if your notions are wrong rather than the data. Nobody said reading mobile statistics is easy.

OS Installed Base

Still, global numbers are the easiest to come by. That's why the next table is global again. It shows the installed base of the various OSs as of 2013; that is, the number of smartphone users who currently use a certain OS.

OS	2013	2012
Android	66%	53%
iOS	20%	19%
Symbian	5%	15%
BlackBerry	4%	8%
Windows Phone	3%	2%
Others	2%	3%

Installed base shares of mobile OSs in 2012 and 2013

As you can see, there are still some people carrying Symbian phones, and that's why the time to ditch Symbian WebKit forever hasn't quite arrived yet. Other than that, the table contains few surprises.

Changes in the Device Market

The mobile market is changing very rapidly. If you compare 2009 to 2013, many leading companies, such as Nokia, HTC, and BlackBerry, saw their market shares dwindle to below 5%, while Samsung, which was disastrously behind in 2009, bestrides like a giant the part of the market that Apple has left unclaimed.

Although it's very hard to make solid predictions, the market will probably change again in the next few years. Basically, the rule is that any device vendor that makes a loss will fall out of the race and may be bought by another player. This has already happened to Motorola (Google) and Nokia (Microsoft), and it's likely that it will also happen to

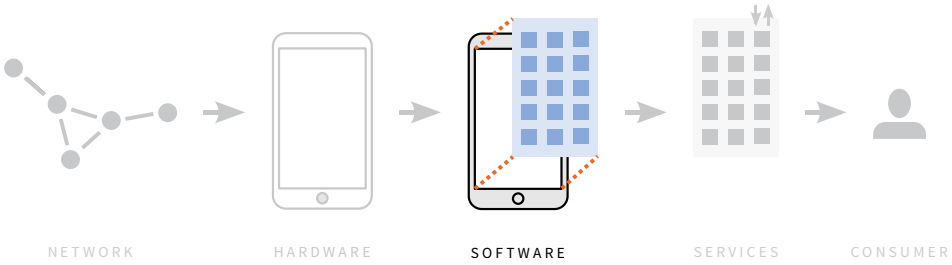
BlackBerry and HTC. Although LG and Sony don't have healthy bottom lines either, they are parts of much larger consortia that may be willing to pay the price of staying in the market. (Entering the market is a lot more expensive than staying in it, despite losses.)

Who's going to acquire them? Motorola's fate gives us a clue. Google sold it to Lenovo, the second-largest phone vendor in China (after Samsung). It's clear that through Motorola, Lenovo hopes to enter foreign markets. In fact, many Chinese companies hope the same. Huawei, Xiaomi, ZTE, Meizu, and others want to become household names outside China. That's already happening in the developing world, but they want to conquer the developed world as well. So it's likely that most of the new players will be Chinese.

Another question is whether more not-really-mobile companies will follow Microsoft's example and buy a device vendor. There have been persistent rumors about Facebook, though I personally believe that it knows it shouldn't go into the hardware business — it's a service provider, and services will trump hardware in years to come. Service providers shouldn't move down-stack to hardware, and neither should OS vendors. They are already in the best part of the market, and owning device vendors won't help them make more money. I can believe in hardware makers being acquired by other hardware makers, but not by anyone else. (And yes, Microsoft is an exception here.)

I'll leave it at that — the market is too volatile to risk specific predictions. I hope this general sketch gives you enough pointers to see the patterns behind future deals and trends in the hardware market.

OS Vendors And Software



Any phone, even a basic one, needs an OS that makes sure the right things happen when the user presses a button or touches the screen. (Please take a moment to recover from this stunning revelation.)

Since 2008, the quality of the OS (the *software layer*) has become more important than the quality of the device (the *hardware layer*). Not all OS vendors understood this — Nokia and BlackBerry, in particular, were too late in adapting to the new order, and suffered as a result. Then again, HTC adapted just fine and is also suffering. A good OS is a pre-requisite for success, but not a guarantee.

Currently, there are two important OSs: Android and iOS. Still, it isn't true that mobile is evolving into a bipolar system. First of all, other OSs still exist — we'll encounter six of them in a moment. Some of them may fail, but we cannot assume that all of them will. Also, the Android ecosystem has grown so huge that it may separate into several branches. We'll get back to that later.

It is unlikely that we'll see more OSs than the eight described below. There may be some adjustments in market shares, but nobody in their right mind will start to build a brand-new OS from scratch. The time-to-market is too long, and nobody is interested in yet another OS.

Incumbents

Let's briefly discuss the eight mobile OSs, starting with the incumbents that actually have market share. We'll leave **Android** to its own chapter because it is so complicated, and continue with the rest.

iOS is simple: Apple holds absolute power and combines hardware and software to create one of the world's most desirable devices. Also, as we saw earlier, both consumers and developers love what Apple has to offer. The only current drawback of iOS is that Apple refuses to produce mid-range phones in the €150–250 class. By now it's clear that this is not going to change, and as a result Apple's smartphone market share will shrink as more and more mid-range Android phones, which are counted as smartphones, will replace feature phones. Still, the current 15% is enough to maintain a healthy ecosystem, especially since the average Apple user is affluent and willing to spend money, and Apple earns rather more than half of the profit in the mobile market.

The **BlackBerry** ecosystem shares the premise of hardware and software created by one company. Unfortunately, BlackBerry ignored the iPhone revolution for too long, and by the time it changed its mind it was too late. BlackBerry 10 is a perfectly fine modern mobile OS, but it was released well after former BlackBerry users had gone over to iOS or Android, and it suffers from a lack of apps. It's likely that BB10 was released too late to help BlackBerry hang on to a piece of the market.

Windows Phone differs from iOS and BlackBerry in that Microsoft licenses it to any interested device vendor. However, Microsoft has strict rules for the hardware (available memory, processor speed, that sort of thing), and vendors are not allowed to change the user interface beyond a few colors. Thus, it is hard to distinguish yourself as a Windows

Phone vendor — especially when compared to Android. Meanwhile, the Microsoft-Nokia deal has made sure that only Nokia remains an important Windows Phone seller. The OS isn't exactly dead, but it needs a major victory in order to capture more than a few percent of the market. Whether the acquisition of Nokia will give Microsoft such a victory is unclear at the time of writing.

Nokia's **S40** OS, though not officially "smart," continues to hold a fat slice of the developing world market, although it's on the defensive against Android-based competitors. S40 is clearly not in the same league as the others, but it has the saving grace of being cheap. Microsoft announced at the time of writing that S40 would be phased out, but it's still popular in poorer parts of the world, where people don't have the money to buy new phones every two years. That makes it likely that S40 devices will continue to be used for a while.

Challengers

Samsung has a history of creating lines of phones that run not on Android but on an OS created in-house. From 2010–2012 the Samsung Wave devices were powered by bada, and when bada folded (for unclear, probably political, reasons) Samsung announced that it would create the new **Tizen** web-based OS together with Intel. bada and Tizen are Samsung's insurance against Google's future plans with Android. If Samsung doesn't like these plans it can always switch from Android to its own OS — or threaten to do so. Meanwhile, the new OS can be tested by a small segment of the Samsung market.

This all sounds great in theory. The problem is that the first Tizen phones have been announced, canceled, reannounced, and again canceled. Right now it is unclear whether Tizen will ever amount to

anything, though a watch running the OS has been released. If, against all odds, Tizen enters the mobile market, it will likely be a hit, since Samsung will use its considerable corporate muscle to make it a hit. So don't write off Tizen quite yet.

Before Nokia decided on Windows Phone it was creating its own new operating system, MeeGo. Once the Microsoft deal went through, development ceased, but a small company of ex-Nokia people called *Jolla* decided to continue the work and create a phone based on MeeGo, now renamed **Sailfish**. At the time of writing it is only available in Finland and China. I have not seen it, so I'm not able to judge its quality. It is possible that Sailfish will become a new player — ex-Nokia people usually have good operator relations, and that counts in the richer parts of the world.

Of the desktop browser vendors, Mozilla was the last to enter the mobile market. In addition to a browser for Android, it decided to create its own web-based operating system, **Firefox OS**. Mozilla aimed for the low-end market, struck deals with operators and device vendors, and went to work. Meanwhile, Firefox OS phones are on sale in Latin America especially, and more releases are planned. So far the result has been modest, but growing a new OS in the face of cheap Android opposition is not easy. It'll probably take until 2015 before we see whether Firefox OS is going to be a major challenger.

The Web as an OS

In 2009, Palm astonished the world by announcing it was going to use the web as its next platform. Native apps would be written in HTML, CSS, and JavaScript, and the OS would be called *WebOS*. The plan was great; the execution lousy. Palm didn't bother to reach out to web developers, and the marketing was a disaster. WebOS disappeared silently.

Despite this initial failure, the web may have a future as an OS. In fact, this is the whole premise Firefox OS is based on, while Tizen is moving in the same direction, and BlackBerry 10 allows HTML5 apps as well. In all cases, you can submit an app created in HTML, CSS, and JavaScript to the app store and have it behave as a native app when it comes to installing and such.

You may have noticed that the platforms supporting HTML5 apps are minority ones. Android and iOS have no HTML5 app capability to speak of — even Chrome apps don't work on Android at the time of writing. The Android and iOS native app ecosystems are thriving, and they don't need HTML5.

Thus, HTML5 apps are weapons in the hands of the losers or challengers in the mobile world. They need a unique selling point, and yet another native app system is not going to cut it, so why not the web? It makes sense from their perspective. Still, I think it's not enough — these platforms will also need the operator relations necessary to bring their devices to the masses. HTML5 alone will not save them.

***

This whirlwind tour of the mobile world has given us a clearer idea of what the differences with the traditional desktop world are. However, so far we haven't found any practical, technical information for web developers. Also, we haven't yet discussed the part of the software layer that's most important to us web developers: the browsers. That's why they are the topic of the next chapter.

Chapter 2

Browsers

Chapter 2

Browsers

If you're used to the simple five-browser ecosystem that exists on the desktop, you're in for a surprise in the mobile market. So far, I have identified about 30 mobile browsers, ranging from lousy to great. Not all of these browsers are equally important: in fact, about 20 of them are somewhat marginal. And just like on desktop, there may be differences between two versions of the same browser.

The Google browsers, Android WebKit and Chrome, come in several flavors, and each flavor may have several versions. In fact, the Android browser situation is so complicated that I'm going to give it an entire chapter of its own. The current chapter mostly ignores Android and instead talks about the other platforms, in particular iOS, as well as some general principles.

You will find no compatibility information here: by the time the book is printed, it would be outdated. You should turn to the companion site at http://quirksmode.org/mobilewebhandbook for details on the differences between the browsers.

Browser Types

There are four browser types on mobile: default browsers, downloadable browsers, proxy browsers, and WebViews. These categories overlap in places: a browser does not necessarily belong to just one category. For instance, the proxy browser Opera Mini is downloaded by many users, but is the default browser on some feature phones.

Default Browsers

Every phone has a default browser; that is, a browser that's part of the firmware, usually developed by the OS vendor. Thus Safari, developed by Apple, is the default browser for iOS; and IE, developed by Microsoft, is the default browser for Windows Phone. The table below summarizes the default browsers of the platforms.

Platform	Default browsers	Remarks
iOS	Safari	
Android	Android WebKit or Chrome	Several flavors of both (see next chapter)
BlackBerry	BlackBerry WebKit	
Windows Phone	IE	
Symbian	Symbian WebKit	
Firefox OS	Firefox	
Sailfish	no name yet	Gecko-based
S40	S40 WebKit on older ones; Xpress on Asha.	Xpress is a Gecko-based proxy browser
Other feature phones	Varies: Opera Mini, NetFront, UC Mini, or others	Opera Mini and UC Mini are proxy browsers.

Sales market shares of mobile OSs in 2012 and 2013

Most default browsers are tightly integrated with the underlying OS, to the point where it is not possible to update the browser separately. Thus, in order to get a new Safari version you have to update iOS; the same goes for IE and Windows Phone. This causes default browsers to develop more slowly than other types of browsers, which could mean in future we have to go through another period where one old, bad default browser holds back the entire mobile web, just as IE6 held back the desktop web. Fingers crossed.

Incidentally, device vendors frequently refuse to give their default browsers names. That's why I use the unimaginative but fairly clear "[Platform] WebKit" when necessary, and my compatibility tables are riddled with Android WebKit, BlackBerry WebKit, Symbian WebKit, and more.

Downloadable Browsers

There are a lot of browsers users can download and install for themselves. Opera, Firefox, Chrome, and UC are a few important ones. In practice, this is only possible on Android, since installing other rendering engines is not allowed on iOS, and no vendor has yet produced a downloadable browser for the small platforms.

One advantage downloadable browsers have over default browsers is that it's possible to update them whenever a new version is available. The latest and greatest features usually land in downloadable browsers first, which is one of the reason web developers tend to like Chrome, Opera, and Firefox. We web developers are not like regular consumers in that respect, though.

It appears that there is a difference between the Western developed nations and the developed nations of east Asia. In the West, few consumers bother to install a different browser — or even know it's possible. In Asia, consumers do download alternative browsers, such as UC or QQ in China, and Puffin in Korea. A common reason is that these browsers offer better integration with local social networks. Asian browser statistics often show downloadable browsers that rarely occur in the West.

What's the point of creating a downloadable browser? The answer is a combination of becoming or staying relevant on mobile, and making money. These two goals are connected: the more relevant you are, the more money you make. Browsers want more market share, and the best way of getting that is to be included as a default browser on some device or another. Before it comes to that, though, these browsers have to show their worth by making a free version available for anyone to try. We'll get back to making money with browsers later in this chapter.

WebViews

A WebView is an OS's browsing interface for native apps. For instance, a Twitter client may call on the platform's WebView to show a webpage when the user clicks on a link in their feed. A game's help pages may be webpages, in which case the game app uses the platform's WebView to display them.

Apple doesn't allow the installation of other rendering engines on iOS devices. Therefore, other browsers wanting to move to iOS are forced to use Apple's WebView. This goes for Chrome on iOS, and also for Opera Coast.

In general, WebViews are separate programs that use many low-level components (such as rendering engines) of the default browser, but may differ in other respects. Testing on WebViews may therefore be a good idea if you expect your pages to run in them.

Proxy Browsers

Then there are the proxy browsers. Their rendering engines, responsible for parsing and executing HTML, CSS, and JavaScript, are found not on the device but on a remote server. They do this to save their users money.

The opposite of a proxy browser is a full browser, and it works as we expect a browser to. When the user requests a page, the browser fetches the HTML, CSS, JavaScript and other assets via HTTP, and once it has everything, it renders and shows the page. All of these steps take place on the client, and take up memory, processor time, and battery life.

Proxy browsers are different:

1. The user requests a page. They send not a normal HTTP request, but a special request to a special proxy server over an encrypted connection.

2. This proxy server makes the normal HTTP request to the web server the user wants to access. It requests the HTML as well as all assets, such as CSS, JavaScript, images and so on.

3. The proxy server contains a rendering engine, which renders the page as usual.

4. The proxy server then compresses the rendered page into a kind of image of the website: think of it as a PDF or an image map. It has hotspots for links, and the user can also select text and zoom a bit.

5. The proxy server sends this file to the client, again over an encrypted connection.

6. The client shows the file to the user. If the user taps on links or does something that requires code execution, the process is repeated.

Opera took the lead in the proxy browsing world primarily because it was the first to see the opportunities and enter the market. Nowadays, though, serious competition is available. There are three important proxy browsers:

1. Opera Mini: used throughout the world, especially in developing countries on low-end devices. Based on Presto at the time of writing, although Opera will eventually switch to Blink.

2. UC Mini: used mainly in China but branching out powerfully across the world. This browser will become more important as time goes by. Based on Gecko.

3. Nokia Xpress: the default browser for Nokia's Asha (S40) low-end phones, and also available for Windows Phone. Based on Gecko. Now that the Asha line is discontinued by Microsoft, it will gradually lose its market share.

Advantage: Cheap

Proxy browsers primarily serve to save the user money. Because all the proxy client has to do is show static files, allow for clicks or taps on links, and generate a simple UI, it's fairly light and able to run even on low-spec phones. Users do not have to buy an expensive smartphone in order to access the web.

Besides, all the client receives is a highly compressed file, which is much lighter than raw HTML, CSS, JavaScript and image files, and it uses only a single request and response. This saves a lot of mobile data traffic — Opera claims up to 90%. Also, this will work even on older networks, which is important to developing-world operators that don't want to spend money on upgrading their entire network.

Thus, proxy browsers serve to make the web accessible even to low-income users who can't afford a desktop computer or a smartphone. Unsurprisingly, they're especially popular in the developing world, while being marginal in developed countries. Still, even affluent smartphone users on excellent connections will notice a distinct increase in speed when they switch to a proxy browser.

Disadvantage: No Client-Side Interactivity

There's a disadvantage to proxy browsing, too: no client-side interactivity. Proxy browsers support JavaScript, but every time the user causes a JavaScript event (by clicking on an Ajax link or something similar), the client sends a request back to the server for instructions. The server executes the script, fetches new assets if necessary and sends back the updated page, which, as far as the client is concerned, is a completely new page.

It's important to realize that this lack of client-side interactivity is a feature, and not a bug. By giving up client-side interactivity, users save themselves a lot of money. Executing JavaScript costs users money, and some prefer not to pay the price.

Working with Proxy Browsers

You must learn to work with proxy browsers. Download Opera Mini to your iOS or Android device now and start testing in it. A proxy browser doesn't quite work like the browsers you're accustomed to, and many users will get their first taste of the web via a proxy browser. Having at least some experience with them is mandatory.

The problem is not in the HTML or CSS — they work pretty much as you'd expect. It's in the JavaScript that you'll encounter the most serious problems. Any time a proxy browser encounters anything dynamic, it has to go back to the server and ask for new instructions. Thus, there's always a lag of a second or more between activation and execution.

Although proxy browsers support JavaScript, most of them disallow certain events. For instance, if you have an `onscroll` event handler, it should fire whenever the user scrolls. But in a proxy browser, that would mean making a server request with every few pixels of scrolling, which would make the page completely unusable. Therefore, proxy browsers disable the `scroll` event. The same goes for the mouse and touch events.

As a rule of thumb, assume that only events that clearly show the user's intent to load new data will work in proxy browsers. In addition, `mouseover` is widely supported because so many websites depend on it, and load and unload because they will be processed on the server anyway. You can expect `click`, `change`, `focus`, `submit` and the like to work, but `mouseout`, the touch events, the key events, `resize` and `scroll` will not work.

I advise you to keep it simple and concentrate on the `click` event, which always works everywhere. Add `submit` if you're working with forms. That's it, though — do not expect other events to work on proxy browsers.

Hybrid Browsers

Since saving bandwidth is such an obviously excellent idea on mobile, the true proxy browsers have been joined by hybrid browsers: browsers that can function either as full or as proxy browsers. In most of them you can switch bandwidth saving on and off. They include Amazon Silk, Puffin, Opera Mobile, and Chrome. Unfortunately the details of their hybrid behavior vary a lot, and it's hard to give general rules.

Exactly how hybrid proxy browsers divide up the work between client and server depends on the browser and the settings. See the Silk description at http://smashed.by/silk; the Chrome data compression proxy description at http://smashed.by/data-compression; and for more information on Opera Turbo http://smashed.by/turbo.
I have not been able to locate similar instructions for Puffin.

The iOS Browser Situation

Now that we know the various browser types, we can understand the iOS browser situation. Remember the crucial fact: Apple does not allow the installation of another rendering engine.

1. The iOS default browser is Safari. Duh.

2. In addition, iOS has a WebView for native apps that need it. Up to and including iOS7 it was slightly different from Safari, but at the time of writing the promise is that these differences will disappear in iOS8.

3. Chrome on iOS may not install its Blink rendering engine, and is therefore forced to use the Apple WebView. The same goes for Opera Coast.

4. Opera Mini, however, neatly evades Apple's restrictions because its rendering engine resides on a server. Installing the Opera Mini client is allowed, and therefore this browser is available on iOS.

In other words, the only non-Safari iOS browsers that it makes sense to test in are the proxy browsers. At the time of writing there's no other proxy browser for iOS but Opera Mini, but that might change.

In particular, Chrome on iOS tests are relatively useless. Although the Chrome app offers you integration with your Google account, when it comes to actually rendering webpages it must use Apple's WebView. Thus, although you can test on Chrome for iOS if you feel like it, this does not tell you anything about the real Chrome on Android, which is a completely different browser.

The Browser Situation On Other Platforms

The other platforms are even simpler to understand than iOS. They have their own default browsers, and usually Opera Mini is also available. Although in general the installation of other rendering engines is allowed, no vendor has yet decided to build a new browser for BlackBerry, Windows Phone, or any of the others.

Rendering Engines

Every browser has a rendering engine that is responsible for the interpretation of HTML, CSS, and the DOM parts of JavaScript. Just like on desktop, there are four important rendering engines on mobile: Gecko, Trident, WebKit, and Blink. In addition, Opera's old Presto engine lives on in Opera Mini for now.

Until about 2010 BlackBerry, NetFront, UC, and a few other browsers had their own proprietary rendering engines, but with the advent of mobile browsing as core platform functionality it became clear that these engines were inferior to the desktop ones, especially in JavaScript and performance. Therefore all proprietary mobile rendering engines were replaced by desktop ones.

Most browser vendors decided to use WebKit. Trident and Presto, back when it existed, were proprietary, and so not an option. As for Gecko, its use beyond Firefox is restricted to UC Mini and several Nokia-descended browsers. The lack of adoption is probably caused by the fact that back in 2009, when most vendors took these decisions, Gecko was still far too heavy for mobile processors and memory constraints.

Meanwhile Mozilla has streamlined its engine in order to create Firefox Mobile, but that change came too late to profit from the initial wave of rendering engine replacements.

Google forked Blink from WebKit in 2013, when the wave of replacements was over. Nowadays it's becoming an option for Android vendors. We'll go into that in the next chapter.

There Is No WebKit on Mobile

So many mobile browsers use WebKit as their rendering engine that it's more efficient to list the ones that do not:

- IE Mobile uses Trident.
- Opera Mini uses Presto, but will eventually replace it with Blink.
- The Chrome browsers use Blink. We'll get back to them in the next chapter.
- Firefox Mobile and Firefox OS use Gecko.
- UC Mini, Nokia Xpress, and the default browser on the Sailfish OS by Jolla also use Gecko.

Any browser not mentioned above uses WebKit. At first sight, the fact that so many browsers use WebKit seems like a powerful aid to web developers. Unfortunately, if a browser uses WebKit it does not mean it's the same as any other WebKit-based browser. In fact, there are considerable differences between them.

WebKit is a rendering engine, not a browser. If you hand it HTML, CSS, JavaScript, and images, it will deliver a rendered page. However, it does not contain the modules necessary to request the assets or to actually show the rendered page on the phone's screen. It depends on the OS

for interfacing with the keyboard, mouse, and touchscreen. Platform owners have to provide all these functionalities.

WebKit provides support for hardware-accelerated animations but does not contain the modules that communicate with the GPU and that make sure that hardware animations actually show up on the screen. If you want modern form fields such as `<input type="date">`, you must write the date interface yourself. WebKit includes Apple's JavaScriptCore as the default JavaScript engine, but you may decide to switch to another, such as Google's V8. Finally, you may use a different WebKit version than the other guy, but even if you don't, two browsers that both use WebKit 537 may be quite different.

So, there is no WebKit on mobile. A lot of browsers use more or less the same rendering engine but differ a lot in their details. Testing your website in all individual WebKit-based browsers is best. If it works in Safari for iOS, it will not necessarily work in BlackBerry WebKit, or Android WebKit, or Obigo, or Symbian WebKit, or Dolphin for Android, or… well, you get the point.

Making Money From A Browser

Why do people make browsers? There are two fundamental reasons: providing your platform with one, and making money. Any smartphone needs a browser. Therefore Apple, Google, Microsoft, Samsung, BlackBerry and others must provide one. Simple.

However, other vendors want to make money with their browsers — even if only enough to pay their engineers. There are three business models for making money from browsers:

1. Selling your company (and browser).
2. Selling licenses for your browser.
3. Search engine deals.

When I started on mobile back in 2009, I tested all the downloadable browsers I could find. Most of them were pretty crappy, but there was one notable exception: the Iris browser for Windows Mobile created by a small Canadian company called Torch Mobile. Several months later BlackBerry acquired the company to build a new WebKit-based browser for its platform, netting the founders and engineers a nice bit of money.

This doesn't happen very often. I had the feeling that back in 2012 the small Californian Dolphin browser groomed itself for acquisition by Facebook, but nothing came of it. Not all that many companies are interested in buying a browser, it seems, and the ones that are have already done so.

Selling licences is a more forward-thinking business model. Opera, especially, makes money from Mini licenses sold to operators, mostly for use on feature phones without a good default browser. The operator gets a customized Opera Mini build for their devices without the Opera logo. This is good for the operators, since browsing users spend more money, but the operators don't have to spend money on creating their own browsers. I assume UC has similar deals in place.

Finally, all browsers have deals with all search engines in which they get a small fee every time a browser user uses the search engine. These deals are shrouded in mystery. The fact that they exist is well known, but the details, especially the financial ones, are secret and will likely

remain so. All browsers do it: it's the easiest way for a browser vendor to make money. The search engine deals are not restricted to downloadable browsers — default browsers do the same, both on desktop and on mobile. The deals are more vital for downloadable browsers, though, which usually don't have other sources of income or the backing of a wealthy corporation.

The search engine deals become more valuable as more people use your browser. It's in the interest of downloadable browser vendors to encourage as many people as possible to use them. Whether they will succeed is an open question.

Statistics

It's time to take a look at statistics again. The best browser market share stats are the ones that come from your client's log files. Study them to find out what kinds of phones are used to visit their website.

Be aware that users of some browsers might not be able to use the website and so might be underrepresented. I usually look at statistics for the homepage or another important landing page and compare them with a few other pages. If a certain mobile browser is visiting the homepage in decent numbers but is nowhere to be seen elsewhere on the website, users of that browser are likely encountering a problem that you must solve.

Finding and using general worldwide mobile browser market shares is fairly hard. What we need are the mobile browser statistics of a first-rank website such as Google or Yahoo. Unfortunately, these companies keep their statistics a secret. As we saw, search engine vendors pay browser vendors a small commission for every query they send, and

they want to hide these vital statistics from their competitors (and from browser vendors). That's why they do not share the browser make-up of their homepage hits.

So, we're reduced to using analytics services that gather these statistics from their clients and share them freely. Unfortunately, these services have a self-selecting bias because site owners (or web designers) have to sign up for them and install a counter script. Thus, even though these services present global data, it comes from a specific subset of websites. I encourage you to sign up the sites you make to one of these services and make the data a little more representative.

The choice is yours, then: either use the statistics, knowing they're incomplete and biased; or use none at all. To me, any data is better than no data, but your mileage may vary. At the time of writing, I know of three such services, and I encourage you to compare them.

- StatCounter (http://smashed.by/statcount)
- NetMarketShare (http://smashed.by/netms)
- Akamai (http://smashed.by/akamai)

Personally, I prefer StatCounter because NetMarketShare puts tablets and mobile devices in one category, and at the time of writing Akamai's "New Features", which comprise most of the mobile data, have serious and persistent interface problems. (You should try it, though. Maybe the problems have been solved by the time you read this.) So I used StatCounter for the data below.

Don't stare yourself blind on tiny differences that are statistically meaningless. What you're after with global stats is the broad picture of who wins and who loses. Chrome is a clear winner (but see the next chapter), while BlackBerry, Nokia, and Opera lose.

Browser	Q2 2014	Q2 2013	Q2 2012
Android WebKit	25%	30%	22%
Safari	23%	26%	24%
Chrome	18%	3%	-
Opera	12%	16%	22%
UC	10%	9%	8%
Nokia	4%	7%	11%
BlackBerry	2%	3%	5%
NetFront	2%	2%	4%
IE	2%	1%	1%
Other	2%	3%	3%

StatCounter: Global mobile browser stats, Q2 of three years

One note: what is "Opera"? Opera Mini, or the full Opera Mobile browser? Unfortunately, StatCounter does not give this information. I assume that 99% consists of Opera Mini, because that would align well with the fact that Opera is mostly present in developing countries, but that's a guess on my part and I may be wrong.

Still, all this information doesn't tell you which browsers will visit your client's site. If you don't have specific stats available, take a look at the stats for your country. They can be very, very different from the global stats. See the next table, for instance.

Browser	US	UK	India	Brazil
Android	21%	17%	12%	31%
Safari	50%	46%	1%	14%
Chrome	21%	19%	4%	37%
Opera	1%	4%	25%	6%
UC	2%	1%	34%	1%
Nokia	-	-	10%	3%
BlackBerry	1%	8%	-	-
NetFront	-	-	7%	-
IE	2%	3%	1%	4%
Firefox	-	-	-	1%
Other	2%	2%	6%	3%

StatCounter: Mobile browser stats of four countries, Q2 2014

Can you spot the differences? Safari rules in the developed West, but not elsewhere. BlackBerry is wiped out, except in the UK. UC is the largest browser in India, while NetFront also retains part of the market. IE and Chrome are more successful in Brazil than in other countries.

As you can see, there is no global mobile browser market — just a collection of local ones.

Although your country's stats are much more useful than global ones, there might still be factors affecting your site that influence the exact browser make-up. But if you don't have stats for that site, you're forced to use country stats instead.

In any case, you should now have some idea of which browsers you need or want to target, even if it's only elaborate guesswork. This will inform your device purchases.

***

Now that we've gone through the simple stuff it's time to look at the more complicated part of the story: Android.

Chapter 3

Android

Chapter 3

Android

As we saw in the Mobile World chapter, nearly four out of five smartphones sold are Androids. Thus Android is the most important mobile OS, but that alone is not the reason it is the sole OS to get its own chapter. The problem web developers face on Android is that the default browser situation is complicated by several factors absent on other platforms:

1. The old default browser, Android WebKit, is in the process of being replaced by Chrome.
2. Both Android WebKit and Chrome come in several flavors, and even if they have the same version numbers one flavor is not necessarily equal to another.

In order to understand why, we have to take a quick look at Android itself before starting on the browsers.

Although this chapter frequently refers to subtle differences between browsers it contains hardly any examples of those differences. As usual, the detailed browser notes can be found at the companion site: http://quirksmode.org/mobilewebhandbook.

Structure and purpose

Google's purpose with Android is to increase the use of its own services. By offering a modern smartphone OS with its own apps and search engine, Google entices more people to use its services, which leads to better data, which leads to more effectively targeted advertising, which leads to more profit. And Google gets a fat slice of the mobile market, which can't hurt.

It made economic sense for Google to provide Android free of charge. In 2008–9, this offer fell on fertile ground. The mobile world was shaken up by the iPhone, and most companies understood they needed a comparable operating system in order to remain relevant in the smartphone business. The Asian vendors, as well as Motorola, were quick to take Google up on this offer. (Nokia and BlackBerry thought they could keep abreast of the market on their own. They were wrong.)

Differentiation

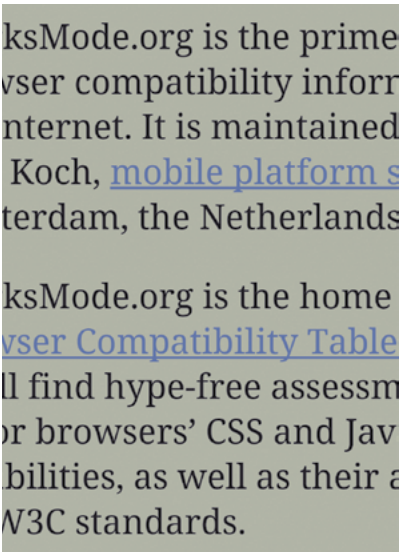
Despite this huge uptake, device vendors had a problem. If consumers can choose between a Samsung Android, an HTC Android, a Sony Android, and a Motorola Android that are all exactly the same, why would they care whether they buy one brand or another? Device vendors wanted to differentiate their devices in the eyes of the consumers.

Since Google wanted them all to adopt Android, it gave (and gives) ample opportunity to differentiate. Device vendors do so mainly by creating feature-rich Android user interfaces: Samsung TouchWiz, HTC Sense, MotoBlur, and all the rest of the UI layers. They're also free to experiment with new features — we'll encounter Samsung's take on touchscreen hover in the CSS chapter.

For reasons of differentiation, device vendors make changes in the default browsers. This is the main reason that Android is more complicated than the other platforms. Even if an HTC and an LG device both run Android WebKit 4.1.1, there will be differences between these two browsers. This is a deliberate attempt at differentiation, and not a bug or oversight.

For instance, one difference between HTC Android WebKit and all the others is that HTC always implements what I call *zoom reflow*. When you zoom in to less than the width of a line, most browsers just show part of the line, requiring you to pan horizontally in order to read it. HTC Android WebKit, however, changes the text width so that it fits in the screen. Apparently HTC doesn't mind the associated processor and battery cost.

Do consumers care? Once, when I was talking about Android browsers in a workshop, an attendee volunteered a story about zoom reflow on his old HTC. He didn't know exactly what was going on or that it was HTC-specific, but he complained that his new Samsung didn't



If you zoom in beyond the width of a line, most browsers show you only part of the line, and you have to pan horizontally in order to read it.

HTC Android WebKit reflows the text so that it fits on the screen.

do it. This is what differentiation is all about: "Oh, my new Samsung doesn't do this cool thing my old HTC did. Better go back to HTC next time." One anecdote does not constitute a robust data set, of course, but it shows what differentiation aims to achieve.

Despite the necessity of giving device vendors the opportunity to differentiate, Google has always been concerned with the unity of the Android platform. App and web developers generally support Google here, since less differentiation makes their lives easier, but device vendors oppose it. It's useful to read Android news through the lens of the tension between differentiation and unification. Once you figure out if the news item would lead to more or less differentiation, you usually understand who supports it and who opposes it.

Android Updates

The glacial pace of distributing new Android versions has become something of a bad joke. Consumers and developers think that as soon as Google announces a new version, their phones will be updated in a matter of weeks. Instead, it is more likely to take six to twelve months, if it happens at all. The reason, again, is differentiation. Device vendors have to test their UI layers and other changes against the new Android version — and when that's done operators have to do the same.

HTC created an excellent infographic that shows all the steps an Android upgrade has to go through before landing on a consumer's phone, and I based this section mainly on that information. See it at http://smashed.by/htcupdate

Say you have an HTC Android phone bought at (and locked by) Vodafone, and Google releases a new Android version. The first question is whether the phone can handle the new version at all. Maybe its hardware is too old, and in that case HTC isn't even going to try to update it.

Let's say the hardware is good enough. Both HTC and the chipset vendor must now make sure the new version will actually work. They test the new Android version, and if they agree it'll work they continue with the next step.

The chipset vendor creates new drivers and optimizes the new version for the specific chip in the phone. Meanwhile, HTC integrates the new version with Sense, its UX layer, and the HTC apps on the device. If that works, HTC can move straight to testing for its own, unlocked devices.

HTC delivers the new Android plus its own changes to the operators, which have to go through the same testing process for their apps and additions. Only when the operators give the green light can the process continue. This is the reason locked devices are generally updated later than unlocked ones. It also explains why sometimes some operators accept the new version but others don't.

Then follows testing, where HTC and the operators test the new version, find bugs and regressions, fix them, and continue until nothing goes wrong. Then the new version has to be certified by regulatory authorities and Google. If Google, regulators, and operators all give the go-ahead, the OTA (over the air) update is prepared. HTC handles this for its unlocked devices; the operators for locked ones.

The consumer receives an update notification only when all these steps have been completed. The process takes longer than the consumer would like, but that's unavoidable. The only thing the consumer can do to speed up the process is buy an unlocked phone.

Google Services

Apple can change whatever it likes to iOS, and push updates whenever it likes. As we've just seen, Google cannot do the same for Android. However, Google's purpose is not spreading new versions of Android as such, but the use of its services made possible by these new versions. That's why Google opted to decouple its services from new Android versions.

The article at http://smashed.by/mwhb3 gives the best overview of Google Services and the reasons why it's important to Google.

Google Services is a collection of important apps, such as Google Play, Maps, YouTube, Google Chrome, and others. These apps are useful in themselves, but the crucial change is that a large number of low-level APIs, such as the Camera UI, the Account Syncing API that connects you to your Google account, and the Maps API are now parts of the Google Play app instead of parts of Android.

Since Play and all the others are apps, they can be updated independently of Android. Even better: a Play update also delivers new versions of the low-level APIs. This change took place in Android 4.3, and it made Google's services effectively independent of the slow churn of formal Android updates.

Device vendors can accept or reject Google Services. However, they accept or reject the entire package — there is no picking and choosing. If they reject Services, they'll have to provide an alternative app store, maps, video service, and also a browser. Google Chrome is part of Services and is thus available on all Android 4.3+ devices that support Google Services. However — and this is a crucial and often misunderstood point — **Google Chrome is not necessarily the device's default browser**. We'll get back to this — oh boy, will we!

Most vendors have opted in to Google Services. The most important exception is Amazon. Amazon is a Google competitor on service level, so it's easy to see why it wants to use its own services instead of Google's. Among other things, that means it has to provide its own browser, Silk.

Android Browsers

With the background out of the way, let's focus on browsers. The situation is complex, since device vendors still want differentiation, while Google is trying to replace the old Android WebKit browser with Chrome.

Android WebKit

Let's start at the beginning. A smartphone OS needs a browser, and therefore the original Android supplied its own WebKit-based browser. It was not given a name, but I call it "Android WebKit". In some articles and books it's called the "stock" or "default Android browser".

Android WebKit is not Chrome: it has a completely separate codebase that contains completely separate bugs. It still uses WebKit, not Blink, as its rendering engine. Always be careful to distinguish between Android WebKit and Chrome. Fortunately that's easy: any Chrome browser has `Chrome` in its UA string, while any Android WebKit browser does not.

Initially, all Android devices had Android WebKit as their default browser. It is tightly woven into Android, and can only be updated by updating Android itself. Major changes took place between 2.3 and 3.0 and again between 4.0 and 4.1. Minor changes happen with every update. The last Android WebKit version is 4.3. Google does not update or support it any more because it wants to push Chrome instead.

Versions are only part of the story, though. Android WebKit contains a lot of switches that turn certain functionality on or off, and device vendors can set these switches to whatever they like. Since they wanted to differentiate themselves, they did so enthusiastically. Earlier in this chapter we encountered the zoom reflow example, where HTC engaged one switch that the other vendors ignored. Similarly, you'll find the occasional difference between, say, a Samsung Android WebKit 4.1.1 and a Sony Android WebKit 4.1.1. That's why it's important to have Android WebKits from different vendors to test in.

> The differences between the Android WebKit flavors are usually rather subtle. For example, the CSS `min-width` declaration is fully supported only by the Samsung and Xiaomi versions of Android WebKit. All the others may have slight bugs — that depends on how you use `min-width` in your site. Another good example can be found at http://smashed.by/aligntest. It turns out that some, but not all, Android WebKit 4 browsers have a complicated bug when you use a `text-align` of anything but `left` combined with a `direction` of anything but `ltr`. This bug is not likely to ruin your day since the circumstances are so unusual. Still, it proves that one Android WebKit 4 is not the same as another.

Unfortunately, Android WebKit was falling behind the competition, and Google decided to replace it with Chrome. However, this replacement process is anything but straightforward.

Chrome

Google Chrome was unveiled in 2008 as a desktop browser for Windows, Mac, and Linux. Like Safari and a growing crop of mobile browsers, it was based on WebKit. Nowadays it's the most-used desktop browser in the world. In 2012 Google created an Android version, too, initially as a downloadable browser. In 2013, Google split off Chrome from WebKit and created its own rendering engine, Blink. Initially, the differences between the two were negligible, but as time progresses WebKit and Blink will grow apart more and more.

Let's take a brief look at the technical stack. Blink is the rendering engine for HTML and CSS, and it's usually coupled with the V8 JavaScript engine. Blink is very tightly integrated with Chromium, Google's open-source browser that anyone can download and change. Chromium exists for Windows, Mac, Linux, Chrome OS, and Android. Google Chrome is Google's own implementation of Chromium.

This is important because more companies than Google use Chromium, and therefore Blink. Opera is the best-known example: in 2013 it retired its own Presto rendering engine, took Chromium, created its own interface, and released the resulting browser as Opera 14. Russian search giant Yandex also took Chromium and created the Yandex browser. Although these browsers don't have a large market share, they illustrate the point that anyone can use Chromium to build their own browser — and that very much includes mobile device vendors.

It's this browser that Google wanted to push as a replacement for Android WebKit. That's great news for web developers: Chrome is a lot more capable than Android WebKit. It's also good for Google: Chrome gathers user data that Google uses to power its ads. However, device vendors were less enthusiastic. They prefer to capture user data themselves, and they also want to continue to differentiate themselves from the other vendors.

Samsung Chrome… and Others

Now we get to the complicated part. Google made the installation of Google Chrome mandatory for all devices that use Google Services. However, nothing prevents the device vendors from using another browser as their default. For instance, at the time of writing HTC still uses Android WebKit as the default browser for its newest devices.

Samsung chose a different path. From the Galaxy S4 (released in 2013) on, Samsung uses its own default browser, which is a Chromium-based one. I call it Samsung Chrome to distinguish it from Google Chrome. In the previous chapter we saw that Chrome has a browsing market share of about 18%. It's likely that most of this 18% is actually Samsung Chrome, and not Google Chrome.

But that's not all. Since Samsung is required to install Google Chrome if it wants to use Google Services, the S4 and newer devices have two Chromium-based browsers installed: Samsung Chrome and Google Chrome. The same goes for all the other vendors. For example, the HTC M8 (released in 2014) also comes with Google Chrome, even though Android WebKit is the default browser. All non-Google Android 4.2+ devices I tested have Google Chrome installed as an app in addition to their default browsers.

Is Samsung Chrome the same as Google Chrome? By now you won't be surprised to hear that the answer is no. First, Samsung Chrome is frozen at version 28, and it is updated only together with a system update — it's a typical default browser in that respect. In contrast, at the time of writing Google Chrome is at version 36 and can be updated independently of the OS.

Second, I compared the previous Samsung Chrome, which was at version 18, with Google Chrome 18, and found one difference: it did not support `border-radius`, while Google Chrome 18 does. Samsung does support `border-radius-top` and such, though. One wonders what they were thinking. In any case, Samsung Chrome 18 is not the same as Google Chrome 18. (I'd love to compare Samsung Chrome 28 with Google Chrome 28, but that's not possible because Google doesn't have an archive of old Chrome versions.)

It's possible that other device vendors will follow Samsung's lead. Differentiating your browser from the other guy's remains popular among device vendors. They will not stop doing it just because Google politely asks them to. So be prepared for an HTC Chrome, a Sony Chrome, an LG Chrome, and so on.

Also, as we saw, Amazon rejects Google Services and therefore has to create its own browser. The newest version is a Chromium-based one, marketed under the name Silk. (Older Silk versions used WebKit.)

At the time of writing Google Chrome is the default browser **only** on Google Nexus devices and on Motorola devices from the time they were owned by Google. You'll often hear that it's the default browser from Android 4.2 or 4.3 onwards, but this is **not true**. I have never yet

encountered a non-Google device that has Google Chrome as its default browser. The other device vendors like differentiation too much.

In other words, Chrome is falling apart into several branches. At the time of writing it seems that these branches, while not exactly the same, will resemble one another more than Android WebKit branches, but I'm not sure if that will continue to be the case. This is one more thing to keep track of.

The upshot of this is that knowing a browser is Chrome is not enough. You also have to find out which Chrome it is. Finally, using Google Chrome for testing will not guarantee that your website works on Samsung Chrome. It's likely that it does, but not certain.

The Current Default Browsers

This is really complex. Let's try to instill some sanity into the Android browser world by making a list. Here is the situation at the time of writing:

1. Android 2.x devices use Android WebKit as their default browser. Updates occur only with Android updates. Chrome is not available for Android 2.

2. Google's own Nexus and Motorola devices use Google Chrome as the default browser. This browser is an app that can be updated independently of the OS.

3. From the Galaxy S4 on, Samsung uses its own Samsung Chrome as the default browser. Updates occur only with Android updates.

4. Most other Android 4 devices still use Android WebKit as their default browser. Updates occur only with Android updates.

5. Android WebKit is not available on Android 4.4 any more. Device vendors wanting to update to 4.4 have to make a decision about their default browser. Android WebKit? Google Chrome? Their own Chromium-based browser? Something else entirely, such as Firefox? Keep close watch on this: it'll determine the future of Android default browsers.

6. Amazon must use its own browser, since it opted out of Google Services.

Downloadable browsers

The list above only includes the default browsers: the browsers installed at the moment the consumer buys the device. However, our Android browser overview is not yet complete, since we're still missing the downloadable browsers.

We already talked about downloadable browsers in general in the previous chapter. Here the main point is that Google Chrome is effectively a downloadable browser on most modern Android devices, and not the default browser.

The question is whether the average Android user will download any browsers — or even notice Google Chrome is installed. Web developers sometimes think they do, because Google Chrome and Firefox are better browsers than Android WebKit. I have found no evidence yet that the average consumer is aware of that fact. I suspect that consumers will just use whatever browser is visible on the home screen.

The danger is that after testing in a downloaded Google Chrome, web developers may think they've covered all modern Android devices. That's not true. Your downloaded Google Chrome only tells you something about Google devices and other downloaded Google Chromes, and not about Samsung Chrome or Amazon Silk or possible future Chromium-based browsers.

Which Browsers Do I Test In?

Phew. That was complicated. Let's wrap things up with a nice list of browsers you need to test your websites in. There are three required browsers, and a slew of optional ones. The three required browsers are:

1. Android WebKit 4, on a healthy mix of devices and Android versions. A major device lab needs about six to eight of them — one from each vendor. A smaller lab will restrict itself to two or three: Samsung, HTC, and one other.
2. Google Chrome. Download it to one of your Androids if it's not already on it.
3. Samsung Chrome. You will have to buy a high-end Samsung phone released in 2013 or later — most likely a Galaxy S4 or higher.

Identify the default browsers of all these devices carefully. That's pretty simple: look at the UA string (user-agent string; `navigator.userAgent`) and see if it contains `Chrome`. If it does, it's Chrome (though not necessarily Google Chrome); if it doesn't, it's Android WebKit.

Depending on your target audience and your client's log files, you may want to test on the following browsers:

1. Android WebKit 2, again on a healthy mix of devices.
2. Amazon Silk. You will have to buy an Amazon Kindle Fire or later.
3. Other browsers: notably Firefox, Opera Mobile, and UC. Download them to one of your Androids.

Despite all the complexity we encountered, you will find that successful web development on Android is mostly a matter of knowing that these different browsers exist, testing carefully, and doing so on many devices. There will be subtle differences, but most of these browsers support most CSS and JavaScript fine.

If you are developing a very complex or JavaScript-heavy application, however, Android WebKit 2 will become a challenge. It is simply not a very good browser by modern standards — comparisons to IE6 are apt. Cover part of your wall with soft padding so you can throw the device in despair without actually damaging it.

At http://smashed.by/mwhb4 you will find recent figures for Android version market share. These are based on visits to Google Play, and not on browsing, but it will tell you which Android versions are still being used actively.

Be sure to follow the Android device, version, and browser markets carefully. New browsers may appear, while older ones may disappear. In particular, test the default browser of any Android device you acquire very carefully, and do not assume anything. In particular, remember that just because the browser sports `Chrome` in its UA string, this does not mean it's Google Chrome.

Yes, it's complicated. But it's not overwhelming. Keep calm, carry on, and identify each browser carefully.

***

Now that we've exhausted the mobile browser market it's time to turn to the nuts and bolts of mobile web development. The next chapter will treat the viewports, what they are, and why mobile browsers need three of them.

Chapter 4

Viewports

Chapter 4

Viewports

If there's one thing everybody intuitively grasps about the mobile web it's that mobile screens are far smaller than desktop (or tablet or TV) screens, and that an interface designed for desktop won't necessarily work well (or at all) on mobile. We found that responsive design helps us a lot in solving that problem. Here's a typical example:

```
<meta name="viewport" content="width=device-width,
       initial-scale=1">
@media screen and (max-width: 480px) {
       // styles for screen sizes up to 480px
       // or whichever breakpoint you prefer
}
```

I assume you've seen these two bits of code before, and have a rough understanding of how responsive design works. You may not know all the ins and outs, though, and that's what this chapter is going to teach you. It studies the two components of the code example: the meta viewport tag, and width media queries — as well as some other meta viewport declarations and all width-, height-, and resolution-related media

Media queries are specified by the W3C at http://smashed.by/mwhb5. Apple invented the meta viewport, and all other mobile browsers copied it. Unfortunately there is no real specification yet — well, there is the CSS Device Adaptation spec at http://smashed.by/mwhb6, but this document is written in such arcane, rarified language that even I, who know quite a bit about the viewports, can't make heads or tails of it. I hope it specifies what follows in this chapter. It might not, but then that's the W3C's problem, not ours.

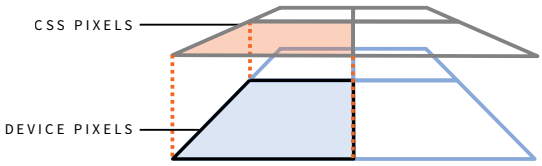
queries. By discussing pixels, viewports, resolutions, the meta viewport, media queries, and related JavaScript events and properties, we hope to gain some insight into how mobile browsers (and we web developers) deal with the fundamental problem of the small screen.

Before we start, let's appreciate the problem mobile browser vendors are facing. Their users expect to be able to visit any site — even those optimized for desktop only. However, such sites are frequently far too wide to be shown comfortably on a mobile screen. Mobile browsers have found a way to display these sites, even though the user experience remains suboptimal. Mobile browsers also offer a way for web developers to adapt CSS layouts to their smaller screens. The ways in which they do so form the topic of this chapter.

Pixels

Before we can investigate the viewports we must say some words about pixels. The humble pixel is the foundation of website layouts, and web developers use it instinctively. Still, there's a lot to know about this fundamental building block. For instance, what exactly is a pixel?

That seems a pretty easy question: a pixel is the smallest area of a computer screen that's able to take on a certain color. The more pixels on a screen, the more you can see at the same time; or, when the device stays the same size but the pixels become more dense, the better the screen shows subtle gradients, and the crisper your website looks.

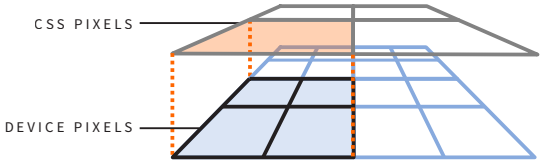


On older screens, and with zoom 100%, one CSS pixel equals exactly one device pixel.

Although all this is true as far as it goes, it's not the whole story. For instance, what exactly happens when you give an element `width: 200px`?

Duh. The element is 200 pixels wide. Silly question.

Sure. But these pixels are not the device pixels on the screen we just described, and an element with `width: 200px` may or may not span 200 of those device pixels. In fact, there are two kinds of pixel:

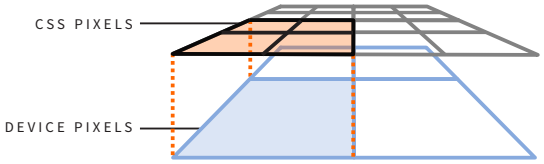


On high-density screens such as Apple's Retina, CSS pixels could span many more device pixels.

1. Device pixels: physical pixels on the device screen, of which there are a fixed amount on any device.

2. CSS pixels: an abstraction layer created specifically for us web developers to be used in our CSS (and JavaScript).

The element with `width: 200px` spans 200 CSS pixels. How many device pixels that equals depends on the nature of the screen (high-density or not) and the zoom factor the user has applied. The more the user zooms in, the more device pixels are covered by one CSS pixel.



If the user zooms out enough, one CSS pixel may become distinctly smaller than one device pixel.

Therefore the element does not necessarily span 200 device pixels. On Apple's Retina screens, for instance, which use twice the pixel density of traditional screens, the element spans 400 device pixels. If the user zooms in it may span even more device pixels.

Still, every CSS or JavaScript test will return an element width of 200px. That's deliberate. While you're working with CSS and JavaScript you don't really care how many device pixels one CSS pixel covers. You gladly leave this complicated computation, which depends on the nature of the screen and the current zoom factor, to the browser. That's how CSS pixels are an abstraction layer created specifically for us web developers. We can just say `width: 200px` without worrying (too much) about what happens on the screen.

Every CSS declaration and nearly every JavaScript property works with CSS pixels, so in practice you'll never use device pixels. The only exception is `screen.width/height`, which is a problem all by itself that we'll return to later.

The Three Viewports

Now we're going to change our CSS to `width: 35%`. Before revealing what happens now, let's add a teaching moment. As soon as any CSS property uses percentages, always ask yourself: "Percentages of what?" You will often find that the answer offers insight to how CSS truly works.

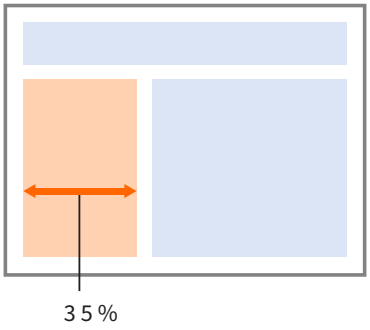
OK, so let's do it. 35% of *what*? Every web developer knows that, on desktop, the answer is 35% of the browser window's width, but not everyone will be completely clear on why this is the case. So it's time for a quick refresher course in basic CSS.

Absent any width declaration in the CSS, each block-level element has a default width of 100%. 100% of what? Every CSS percentage width is calculated relative to the width of the parent element, so the element now takes 100% of the width of that parent element. So what we have here is essentially this:

```
html,body {
       // no width defined; so an implied 100%
}
div.sidebar {
       width: 35%;
}
```

Our `div.sidebar` takes up 35% of the width of its containing block, the `body`. The `body`, having no given width, takes 100% of the width of its own containing block, the `html` element. This element has no given width, either, so it also takes 100% of its containing block.

And what is the `html` element's containing block? Now we have arrived at the **viewport**, which the CSS specification calls the **initial containing block**. This initial containing block is the element that all CSS percentage widths are ultimately derived from and that serves to constrain your CSS layout to a certain maximum width. (You can break out of these constraints by assigning very large widths to elements, but let's simply ignore that for now.)



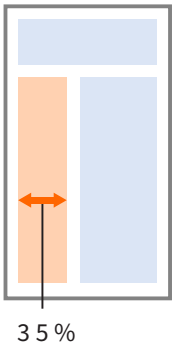
On desktop, the sidebar with `width: 35%` takes up 35% of the viewport width, which is equal to the browser window.

On the desktop, the viewport is exactly as wide as the browser window. Therefore, leaving aside margins and padding, the `html` and `body` elements are also as wide as the browser window. That's why the sidebar takes up 35% of the width of your browser window. This is not particularly groundbreaking news, but you need a clear picture of the mechanism in order to understand what follows.

The Layout Viewport

The problem on small-screen mobile devices (and even on most tablets) is that making the viewport equal to the browser window width would have some very ugly consequences. Mobile or tablet browsers generally have about 240–640 pixels of screen width at their disposal, and an average desktop-only site assumes at least 800px, and preferably 1,024px. As a result, our desktop-designed sidebar with a width of 35% will look horribly squished on mobile.

Some sleight of hand is clearly necessary here. Mobile browser vendors had to make sure that our sidebar displayed relatively well despite the narrow screen. That's why they made the viewport significantly wider than the device screen, so that sites would look roughly as intended. The most common viewport width is 980px, although you can find anything between 768 and 1,024.

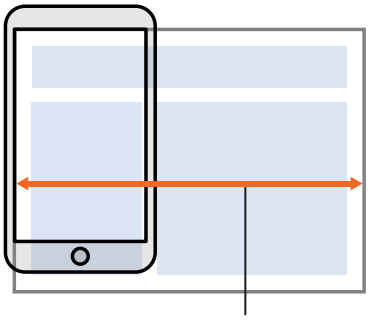


If the narrow mobile screen were also the viewport, sites would get squished horizontally, which is usually quite ugly.

It's likely that a desktop-based site will display well in an 768–1,024px viewport. The element with `width: 35%`, and in fact all elements in the site, will display roughly as a desktop designer intended. Thus the mobile browser can also handle desktop websites, and users will be happy.

There's a trade-off, though. If a mobile browser encounters a non-mobile-optimized site, it zooms out as much as possible in order to give the user an overview. This is not good for legibility, but if you want to display desktop-optimized sites on a small mobile screen something has to give way.

On mobile, then, the viewport is not tied to the width of the actual mobile browser screen any more, but is fully independent. We call it the **layout viewport** — the viewport relative to which the CSS layout is calculated, and which constrains that layout.

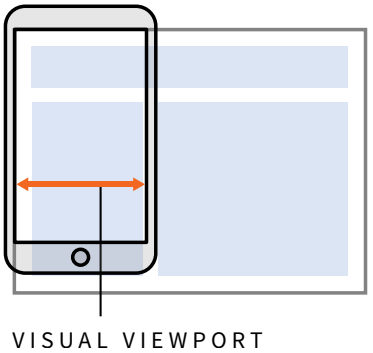


On mobile, the default layout viewport is significantly wider than the screen in order to accommodate desktop-optimized sites.

The Visual Viewport

Although the creation of the independent layout viewport helped a lot with porting desktop sites to mobile, we cannot entirely ignore the screen size of the mobile device. Some CSS declarations have a relation with what the user sees, and not with CSS's fairly abstract initial containing block. Also, occasionally it's useful for web developers to find out how much of the site the user is currently seeing.

So it's time to introduce the **visual viewport**: the area of the site the user is currently seeing. The user can manipulate the visual viewport by zooming out or in, without affecting the layout viewport, which retains its given width.



The visual viewport is as wide as the device screen, and changes in size when the user zooms.

In general, the visual viewport is not all that important to web developers, but if you desperately need to know which part of the site the user is currently viewing, it's there, and you can access the data with JavaScript. as we'll see later.

The Ideal Viewport

By default, a mobile or tablet browser's layout viewport is 768–1,024 pixels wide. Although that saves desktop websites from being squished, it's not ideal, especially for mobile users, since a narrow device screen would be better served with a narrow site.

In other words, the default size of the layout viewport is not the ideal size. That's why Apple, followed by all other browser vendors, introduced the **ideal viewport**, which is the size of the layout viewport that is ideal for the device. Websites shown in this ideal viewport have the optimal width for browsing and reading, and initially the user does not need to zoom.

Still, this ideal viewport should only be used if the site is ready for mobile. That's why it is only implemented if you actively order the browser to do so by including the meta viewport tag. If there is no meta viewport, the layout viewport remains at its default width. The ideal viewport only comes into play when you explicitly call on it:

```
<meta name="viewport" content="width=device-width">
```

This line of code tells the browser to make the layout viewport match the ideal viewport. We'll discuss the details of the meta viewport later in this chapter.

The best-known ideal viewport is the early iPhone's 320×480px, which was upgraded to 320×568px in the iPhone 5. Of course, not many other devices use the same ideal viewport, and that's logical given that most devices have a slightly different physical width or device pixel count. Android phones, in particular, have widely ranging ideal viewports, varying (in my collection) from 320×427px for the Samsung Galaxy

Pocket to 400×600px for the Samsung Galaxy Note I, while most other Android devices use 360×640px.

Defining the ideal viewport is a job for the browser, and not for the device or operating system. Thus, different browsers on the same device may have different ideal viewports. For instance, where the default Android WebKit on the Samsung Galaxy Pocket has a 320×427px ideal viewport, Opera Mobile 12 on the same device uses 240×320. However, the browser's values also depend on the device it runs on. Chrome 34 on the Samsung Galaxy S4 has an ideal viewport of 360×640, but on the Nexus 7 it's 601×962. The reason is obvious: the Nexus 7 is a tablet that has a physically wider screen, and thus the ideal viewport should also be wider.

320×427 and 601×962 may seem to be weird values, and so is the BlackBerry Z10's 342×570, but there are no wrong values for the ideal viewport. It can be anything the browser vendor feels is appropriate to the device. I studied about 50 browsers, and all of them had reasonable ideal viewport dimensions for the device they run on.

The ideal viewport width changes with the device orientation: Chrome/Nexus 7 has an ideal viewport width of 601px in portrait mode and 962px in landscape mode. Later we'll encounter some Safari problems in this area, as well as the way to solve them.

Although it may seem that this huge amount of ideal viewport dimensions makes your job more difficult, this is not the case. You should just tell the browser to use its ideal viewport, which will always give a good result. Then you should use media queries to make your site layout respond to whatever value the browser deems correct. We'll come back to media queries later.

Which Viewport?

Let's quickly repeat our findings:

1. On desktop browsers, the browser window is the viewport (also called the initial containing block), which constrains the width of your CSS layout. It also defines what the user can see.
2. On mobile, the desktop viewport has been split into two: the layout viewport to constrain your CSS layouts; and the visual viewport to define what the user can see.
3. Mobile browsers also have an ideal viewport, which gives the ideal dimensions of the layout viewport for this specific browser on this specific device.
4. It is possible to set the dimensions of the layout viewport to those of the ideal viewport. In fact, this is the basis of responsive design.

Although the concept of a viewport is nothing new, splitting it into three is a recent innovation. That's why you'll often find "viewport" in articles or even W3C recommendations without the author specifying which one. Usually it's pretty clear from the context which viewport is meant, but from time to time you encounter edge cases, or instances where the writer is unaware of the existence of more than one viewport.

Therefore, if you encounter "viewport" anywhere, get into the habit of asking yourself which viewport. Sometimes the answer may be surprising, or completely unclear. In the latter case, don't hesitate to ask the writer for clarification. To get you in the mood, here are two examples from CSS. Both depend on the viewport — but which one?

1. `position: fixed`. The specification says: "The box [with fixed position] is fixed with respect to the viewport and does not move when scrolled."

2. The vw and vh units are percentages of the viewport; `width: 25vw` means the width of the element is 25% of the viewport width.

We'll get back to these two examples in the CSS chapter. For now, try to figure out which viewport they depend on.

Zooming

A few words on zooming are necessary, since it works quite differently on desktop and on mobile. Not only are there technical differences, but also differences in why and how often people use zoom.

On the desktop, zooming typically occurs because of a combination of poor eyesight and tiny fonts. A user struggles to read the text, and increases the zoom level a bit. This will likely happen once, when the user first sees the page. After that there is no more need for zooming, and some modern desktop browsers even remember the preferred zoom level and apply it automatically when the user visits the site again. So on desktop, zooming is typically a one-off occurrence.

On mobile, zooming is very different. First of all it might not be necessary, if the web developer has taken care to create a responsive site or a separate mobile site. (Even if you have, it's useful to still allow zooming, since some users will not be able to read your fonts even in the mobile-optimized site.)

If a mobile browser encounters a non-mobile-optimized site, it zooms out as much as possible in order to give the user a good overview. That's useful, since the user can now choose which parts of the site to interact with. Such interaction involves zooming in on part of the site, and when users want to go elsewhere they could zoom out and then zoom in on the next part. Thus, on mobile, zooming is an iterative process, and it is much more important to the flow of user interaction than on desktop.

What is Zooming?

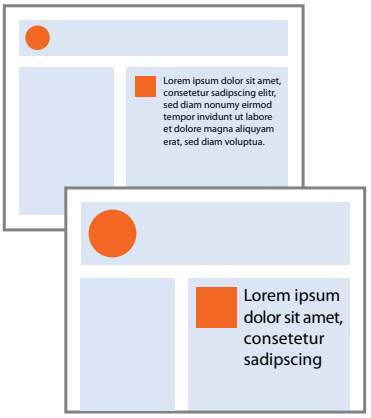
Technically, zooming in is the process of enlarging CSS pixels, typically to arrive at a readable font size. Yet zooming influences all elements on a page. Our element with `width: 200px` still spans exactly 200 CSS pixels, but since the size of those pixels has increased, it now spans more device pixels. Zooming out does the exact reverse: the size of the CSS pixel is decreased and the element spans fewer device pixels.

Thus zooming influences the size of the (visual) viewport on both desktop and mobile. Zooming in makes it smaller, since fewer CSS pixels fit on the screen, and zooming out makes it larger, as more CSS pixels fit on the screen. The zoom factor and the visual viewport therefore have an *inverse relationship*: the larger the zoom factor, the smaller the visual viewport.

The big difference between desktop and mobile is that on mobile the layout viewport is not affected by zooming, but on desktop it is, since it is equal to the visual viewport and it's impossible to change the one without changing the other.

Page Zoom

This may be difficult to picture, so let's use a practical example. We start with *page zoom*, which is the way desktop browsers zoom. We will concentrate on our old friend, the sidebar with `width: 35%`.



The viewport becomes smaller when the user zooms. This causes the CSS layout to be recalculated.

Let's say a desktop browser window is 1,024px wide, which means 1,024 CSS pixels fit on the screen. Furthermore, the zoom level is 100% and the screen has a device pixel ratio of 1 (just nod wisely; we'll cover DPR later), so that one CSS pixel now exactly covers one device pixel.

How wide is the sidebar? 35% of 1,024 equals 358.4, rounded to 358. The element is 358 CSS pixels wide, and because of the specifics of this screen and zoom, also 358 device pixels wide.

Now the user zooms in to 200%. The CSS pixels double their width, which makes the viewport shrink to a width of 512 CSS pixels; at this point, one CSS pixel is as wide as two device pixels. Since the viewport width changed, our element's width is recalculated. It is now 35% of 512, which equals 179.2 CSS pixels wide, rounded to 179.

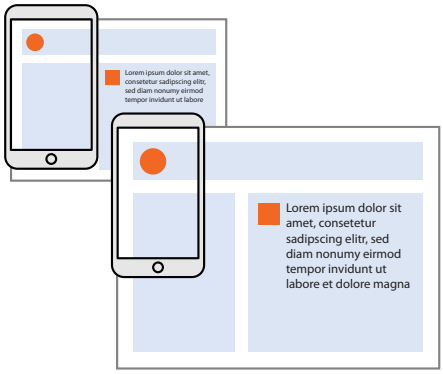
However, since one CSS pixel spans two device pixels in each dimension, it's actually still 358 device pixels wide. That's as large as it was before the zoom — in device pixels.

The font size is the major difference now: a 16px-wide character will now span roughly 32 device pixels, so fewer characters now fit on one line.

Pinch Zoom

Let's do the same on a mobile device. Here it's called *pinch zoom*, and it's fundamentally different from page zoom.

Let's say the mobile browser's layout viewport width is again 1,024px, but the screen is only 320 device pixels wide. The element with `width: 35%` is still 358 CSS pixels wide: the calculation is exactly the same as on desktop. It's now wider than the visual viewport and sticks out of the screen. No matter, the user can always zoom — in fact, this is the exact use case that mobile zooming is trying to solve.



Only the visual viewport changes size when the user zooms; the layout viewport remains the same. The CSS layout is not recalculated.

Now let's again say the user zooms in from 100% to 200%. Again the CSS pixels increase in size, until only 160 of them fit on the screen. However, here the layout viewport stays at 1,024px, so our element doesn't change size: it's still 358px wide and is now definitely much wider than the 160px visual viewport. But the user can always zoom.

Mobile zooming does not cause the CSS layout to be recalculated. Since zooming happens a lot on mobile, and mobile processors are slow and their batteries can drain quickly, not recalculating the layout has definite performance advantages.

Minimum and Maximum Zoom

How much can the user zoom? On mobile, it turns out that it's from about 20% to about 500% — a factor of five, in other words. By using the proper meta directives, which we'll explore later in this chapter, you can widen this range to a factor of ten: 10% to 1,000%. Android WebKit is different, though: it can zoom by a factor of four: 25% to 400%, meta directive or no meta directive.

Wait a minute! What are all these percentages? 10% and 20% and 500% and 1,000% of *what*? Oh good, you remembered the right question. Now answer it — guess, if you must. And don't look at the next paragraph just yet.

If you think these zoom factors are relative to the visual viewport size, you're wrong. If you think they're relative to the layout viewport size, you're also wrong. Browsers calculate their zoom level relative to the size of the ideal viewport. And in case you're wondering, no, this doesn't make the slightest bit of sense.

Suppressing Zoom

It is possible to suppress the user's zooming ability using this meta tag:

```
<meta name="viewport" content="user-scalable=no">
```

However, suppressing zoom is evil. Not flawed, not stupid (well, that too), but unmitigated, inexcusable evil; Sauron-like depths of evil.

A friend of mine is a doctor. One day she was at the top floor of the hospital when her pager bleeped and she was urgently called downstairs for a resuscitation. (Her pager? Yes. The hospital hasn't yet figured out that they could also use mobile phones for such things. But that's another story.) While waiting for the lift to take her ten stories down she decided to briefly go through the resuscitation protocol on an app she'd recently purchased. The crucial scheme that showed all the steps was a bit too small, however, and she tried to zoom in.

She couldn't. It turned out some idiot app designer had turned off zooming; apparently, it was "not necessary." Thus a doctor was unable to view the steps that could save her patient's life because some silly designer's so-called creativity couldn't handle the threat of zooming. That's what I mean by evil. If people zoom in on your carefully crafted page, it means they can't make out a few details. That's your fault, and not the users'. So don't punish them for it.

Chrome for Android allows you to turn off zoom suppression, and that's a setting I check immediately. Other browsers may do the same — I admit I didn't test them all. The takeaway here is that, apart from being evil, suppressing zoom is pointless because users can (and, I expect, will) override it. So don't bother.

Other Forms of Zooming

There's a third form of zooming, and it exists only on Firefox and Safari for the desktop, and on no mobile browser. It's *text zooming*, which means that when the user zooms the font size is upped, but the rest of the page stays as it is. Although this is not a mobile function and falls outside the scope of this book, it's still important for when we discuss em-based media queries below.

Finally, there is the possibility that the user explicitly sets the default or the minimum font size. Usually the default is 16px, but many browsers allow the user to set it to another value. This is not zooming in the strict sense because the user does not use the zoom interface but instead goes to the settings and likely changes the font size once and forever.

Resolution

Resolution is a complicated topic because it has two meanings. On the one hand there is the physical dots-per-inch count of specific devices, and on the other hand there is something called *resolution* in CSS and JavaScript that seems to be the same but is in fact something quite different.

Physical Resolution

All screens have a physical resolution. Dividing the number of pixels by the width of the screen in inches gives you the device's *dots per inch* (DPI for short). More pixels per inch is good, since it means a crisper display. That's why a device's DPI has become an important unique selling point that's touted in every device description.

It is impossible for web developers to know this physical resolution because browsers simply do not have the information available. A few expose the number of device pixels in `screen.width`, but this is not reliable across browsers, and in any case the physical size of the device is not available to JavaScript. Using a device database like WURFL or DeviceAtlas is the only option for web developers who need the physical device resolution. We'll get back to device databases in the Becoming A Mobile Web Developer chapter.

Device Pixel Ratio (DPR)

JavaScript has a `window.devicePixelRatio` property, and CSS has `device-pixel-ratio` (WebKit-based browsers) and `resolution` (all other browsers) media queries, but they have nothing to do with physical resolution. Instead, they give you the ratio of the number of device pixels to the ideal viewport size.

Early iPhones are 320 device pixels wide, and their ideal viewport is also 320 pixels wide. Therefore their device pixel ratio (DPR) is 1. Later iPhones have 640 device pixels, but their ideal viewport is still 320 pixels, and their DPR is thus 2.

DPR does not need to be an integer. We saw that Android WebKit on the Samsung Galaxy Pocket has an ideal viewport width of 320, just like the iPhone. However, the device has only 240 device pixels, and therefore its DPR is 0.75. The BlackBerry Z10 has 768 device pixels, and its ideal viewport width is 342, which gives it a DPR of roughly 2.25.

Web developers use DPR to decide whether or not to send high-resolution images or not. If the device has more device pixels available for each CSS pixel, a high-res image makes sense because it makes better use of the device's capabilities and will please the users.

However, implementing this correctly is technically tricky for reasons that fall outside the scope of this book. Some of the best minds in modern web development have united in the Responsive Images Community Group (http://responsiveimages.org/) to work on a solution, and if this problem interests you, you should follow them.

Like with the ideal viewport, none of these values are wrong, though some appear weird when you see them for the first time. Browser vendors decided on an ideal viewport width that works on the device, and the DPR logically follows from that.

dppx and dpi

The implied unit for JavaScript `window.devicePixelRatio` and the `device-pixel-ratio` media query is dppx: dots per pixel. Actually appending the unit is not allowed, so this is the proper syntax:

```
if (window.devicePixelRatio >= 2) {
       // DPR at least 2. Do something.
}

@media all and (-webkit-min-device-pixel-ratio: 2) {
       // DPR at least 2. Do something.
}
```

The `resolution` media query, though, does require a unit, and the problem is that while dppx is available in most browsers, it is not supported by IE11 and below. Therefore we have to use the dpi unit instead. Since one inch is defined as 96 pixels in CSS, 1dppx is equal to 96dpi. To make the media query above fully cross-browser we have to do the following:

```
@media all and ((-webkit-min-device-pixel-ratio: 2),
              (min-resolution: 192dpi)) {
       // DPR at least 2. Do something.
}
```

Don't make the mistake of thinking physical inches are involved in the dpi unit. Otherwise, it's ready for use.

The Meta Viewport

The main purpose of the meta viewport tag is to match the layout viewport size to the ideal viewport size. It was invented by Apple, and the other mobile and tablet browsers copied most of it. Desktop browsers do not support it, nor should they, since they lack the concept of an ideal viewport. IE is a special case: on phones it supports the meta viewport tag, but it's better to use `@-ms-viewport`. We'll get back to that after the description of the tag.

The meta viewport tag should be placed in the `<head>` of the HTML document and has this format:

```
<meta name="viewport" content="name=value,name=value">
```

Each name/value pair is a directive that gives an instruction to the browser. They are separated by commas. There are five of them:

1. `width`: sets the width of the layout viewport to the indicated value.
2. `initial-scale`: sets the initial zoom factor of the page and the width of the layout viewport.
3. `minimum-scale`: sets the minimum zoom level (how much the user can zoom out).
4. `maximum-scale`: sets the maximum zoom level (how much the user can zoom in).
5. `user-scalable`: prevents user zooming when set to no. This is evil and we will demonstratively ignore it.

Most resources mention a sixth directive, `height`, which sets the height of the layout viewport. Unfortunately the `height` directive isn't supported anywhere at the time of writing, not even in Safari on iOS. (Then why did Apple add it to its documentation? I don't know.)

width

The main purpose of the meta viewport tag is to set the layout viewport to the ideal one. We already saw how this is done:

```
<meta name="viewport" content="width=device-width">
```

Now your webpage has the ideal size for the device it's being displayed on, and the only thing you have to do is find that size via a media query. We'll consider all that later.

The layout viewport width changes when the user switches the device's orientation. For example, Chrome on the HTC One X has an ideal viewport width of 360px in portrait mode, but that becomes 640px in landscape. In general this is what you want: the ideal viewport should respond to the device orientation because landscape offers more width than portrait.

There's one important exception: Safari on iOS. This browser does not adjust the ideal viewport to the device orientation: it stubbornly stays at the portrait values of 320px (iPhone) or 768px (iPad). This is not a bug in the sense that it's an unintended consequence of bad code, but it's still annoying and irregular. My guess is that Apple does this in order to avoid the recalculation of the page layout caused by changing the layout viewport width. That recalculation might cost too much processor time and battery life. (Then again, all other browsers can handle it.) There is a solution to this problem that we'll look at in a moment.

Though the `device-width` value is the correct one for more than 95% of the sites, it is possible to assign another width to the layout viewport. For instance, this gives you a layout viewport of 400px wide in all circumstances:

```
<meta name="viewport" content="width=400">
```

The maximum value browsers support is 10,000 pixels (but why would you want to do that?), and the minimum about 20% of the ideal viewport width. Android WebKit doesn't allow any width value below the layout viewport width. If you specify one, it reverts to the default layout viewport width; usually 980px. IE10 doesn't allow any width above 480px, reverting to the default layout viewport width of 1,024px.

initial-scale

The `initial-scale` directive sets the initial zoom factor of the page. The value 1 means 100%, 2 means 200%, and so on. We've already seen that this zoom factor is calculated relative to the ideal viewport.

Remember: the zoom level is inversely proportional to the visual viewport width. A higher zoom level means a smaller visual viewport. So `initial-scale=1` zooms in until the visual viewport is as wide and high as the ideal viewport. `initial-scale=2` zooms to 200%, so the visual viewport is half as wide and high as the ideal viewport. On the BlackBerry Z10 with an ideal viewport width of 342px, that would be 342px and 171px wide, respectively.

This works fine in most browsers, and it's what one would expect to happen. However, using `initial-scale` has a second effect: it sets the dimensions of the layout viewport to the zoom dimensions as well.

That is, on the BlackBerry Z10 `initial-scale=1` would give a layout viewport of 342×570, just as with `width=device-width`, while `initial-scale=2` would halve that to 171×285.

So it turns out that `initial-scale=1` has exactly the same effect as `width=device-width`. And yes, this is weird. It doesn't make the slightest bit of sense to me.

The Perfect Meta Viewport

An unexpected bonus is that Safari's refusal to switch to landscape width doesn't occur when you use `initial-scale=1`. In portrait mode the layout viewport is now 320px wide, and in landscape mode either 480px or 568px, depending on the iPhone model.

Just to keep you on your toes, IE10 turns out to have exactly the opposite problem: with `initial-scale` it stays at 320px even in landscape mode, but with `width=device-width` it switches from 320 to 480. In order to solve the bug in all browsers it's thus necessary to use the following:

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

Now both the Safari and the IE problems are covered, and your layout viewport responds to the orientation changes of the device. This is the perfect meta viewport, and you should use it in all your projects.

Elements Too Large

Next problem. Suppose you set the layout viewport to the ideal viewport and then add an element that's clearly too wide. What happens when the layout viewport is constrained to 320 (or 360 or 400) pixels, but contains an element that's 800 or 1,000 pixels wide?

Obviously, the element sticks out of the layout viewport. This is common behavior in CSS. Technically, the `overflow` declaration tells the browser what to do now, and its default value of `visible` makes sure the element that's too wide is shown in its entirety and sticks out of its container. So this is no surprise.

But what about the layout viewport? You'd expect it not to react at all to this element, but it turns out that if you use either `width=device-width` or `initial-scale=1` — but *not* both — some browsers stretch the layout viewport to accommodate the element. The compatibility patterns are tricky here, but fortunately the solution is simple: if you use both, most browsers keep the layout viewport intact. This is one more reason to use the perfect meta viewport.

Minimum Layout Viewport Width

The fun part of doing this kind of browser compatibility research is messing things up to see how browsers react. So I decided to give some conflicting orders and see what happened. The results were surprising.

```
<meta name="viewport" content="width=400,initial-scale=1">
```

Now we're telling the browser to set the layout viewport width to 400, and then to set it to the ideal viewport width. It turns out that all browsers react in the same way: they pick the largest available width per orientation. So an early iPhone in portrait orientation would get a 400px-wide layout viewport (the larger of 320px and 400px), and in landscape 480px wide (the larger of 480px and 400px).

Thus you can apply a minimum width to your layout viewport. The example above sets it at 400px, allowing the browser to make the layout viewport larger if device and orientation require it, but not smaller. I'm not sure if there's a practical use case for such a minimum width, but technically it's possible and I encourage you to experiment.

minimum- and maximum-scale

We've examined various aspects of zooming already, but let's repeat it all in the context of `minimum-scale` and `maximum-scale`. These directives allow you to set a minimum and maximum zoom factor. Like `initial-scale`, all zoom factors are calculated relative to the ideal viewport.

Without these directives the browser allows zooming in and out to a factor of five (20% to 500%); with these directives that factor rises to ten (10% to 1,000%). Higher factors are not supported, so `maximum-scale=20` will effectively be `maximum-scale=10`. Android WebKit does not support `minimum-scale`. Also, it zooms to a factor of four (25% to 400%), and it's not possible to change that. IE has some issues with these directives; don't be surprised if they don't work quite like you expect.

@viewport and IE

The meta viewport tag is an odd construct in the sense that it gives orders to the CSS presentation layer but is itself part of the HTML structural layer. Opera proposed a pure CSS syntax:

```
@viewport {
    width: device-width;
    zoom: 1;
}
```

Unfortunately this syntax has not yet been widely picked up. Opera supported it in the Presto rendering engine, but now that it has switched to Blink, Opera has lost it again. The only browser to support it at the time of writing is IE, as `@-ms-viewport`, and it's not exactly the same as the meta viewport tag.

IE supports the meta viewport tag only on phones, and not on tablets. In addition, the tag always uses an ideal viewport width of 320px, because Microsoft wants to stay as close to the iPhone as it can. However, when you use `@-ms-viewport`, IE switches to its true ideal viewport — the one that best matches the device. So the following code gives you a layout viewport equal to the true ideal viewport (for instance, 364 pixels on the Lumia 820), even though the meta viewport tag gives you 320px:

```
@-ms-viewport {
       width: device-width;
}
```

`@-ms-viewport` overrides the tag, so by using both you can make sure that IE takes on its true ideal viewport. In general, this is the best thing you can do since 320px is not always ideal for every Windows Phone device.

Media Queries

We've referred to media queries a lot on this chapter. The time has come to review them systematically. Media queries are nothing but if-statements in CSS. If the width is 800px or larger; if the orientation is landscape; if the resolution is 1.5dppx or smaller: apply these CSS declarations.

There are three categories of media queries:

1. media type queries: what kind of device is this?
2. viewport-related media queries — the meat of this section.
3. feature-related media queries: does the browser support feature X?

> The media queries discussed in this section only work at page level. However, the idea of element media queries is being floated. They would react to the width or height of not the page as a whole, but of a specific element, and would be useful for Twitter or Facebook widgets and such. Although at the time of writing, element media queries are not supported, and there isn't even an agreement on syntax and scope yet, they are a good idea and I hope they'll be implemented eventually.

We're not going to talk about the last category. Although some may be useful for mobile web development, only the useless ones are widely supported, and none of them has anything to do with the viewport.

Media Types

Originally, the idea was that media types would allow you to distinguish between different types of devices. Unfortunately, that idea has failed; the only truly useful media type is `print`. The others have never been implemented properly. For instance, TV browsers should support the `tv` type but don't. Similarly, mobile devices are supposed to react to the `handheld` media type but don't. The reason why they don't is instructive.

Back in the bad old days, when mobile browsers could at most handle WAP and the XHTML-MP subset of HTML, a good many of them followed the spec and supported `handheld`. Web developers eagerly pounced on that to send these browsers much simpler versions of their styles and scripts, since they didn't support the full versions anyway.

On came modern mobile browsers such as Opera, Safari, and BlackBerry, and they saw how web developers were using the `handheld` type. Since they supported HTML, CSS, and JavaScript properly they wanted to get the full styles and scripts. The obvious way of doing that was not supporting the media type, so that's what they did. Thus, the mark of a modern mobile browser is not supporting `handheld`. TV browser vendors made a similar decision.

It's the same arms race all over again: web developers want to distinguish between less capable and more capable browsers; then new browsers are released in what was the less capable category, and to end up on the right side of developers' detection efforts they start fudging their identities. Minor browsers did this back when we thought only Netscape and IE were capable; mobile browsers do it now, and no doubt many more browsers will follow the same path.

So don't bother with types, except for `print`. Print style sheets are very useful and underused. Although they fall outside the scope of this book, I urge you to use them in your projects.

Syntactic Notes

This is a media query. The styles are used when the layout viewport is 400px wide or less:

```
@media all and (max-width: 400) {
       div.sidebar {
              // these div.sidebar styles
              // are used when the layout viewport
              // is 400px wide or narrower
       }
}
```

There are several important points here. First of all, all media queries require a media type, and usually `all` is the best one to use.

Second, you should always use a `min-` or `max-` prefix to your media queries. Usually you're not interested in an exact value, but in a range of values. The example above works whenever the layout viewport is 400px wide or less, and below we'll see an example that should work when the resolution is 1.5 or less.

Finally, the unit of this media query is the pixel, even though no formal unit is defined. You can also use any unit that's valid for a CSS length, such as em or cm, though you have to explicitly define them. Only percentages are not very useful, and to be honest I'm not even sure they're supported. (Besides, percentages of *what*?)

You can use as many queries as you like. The `and` is a logical and, and the comma is a logical or. So let's take a more complicated example. The next media query will apply if the layout viewport width is 400px at most **and** the orientation is portrait **and** the resolution is 1.5 at most. That last condition needs two media queries because of browser incompatibilities, and they're separated by a comma (device pixel ratio 1.5 or less **or** resolution 144dpi or less).

```
@media all and (max-width: 400) and (orientation: portrait)
       and ((max-resolution: 144dpi),
       (-webkit-max-device-pixel-ratio: 1.5)) {
              /* styles for when the layout viewport
              is 400px wide or narrower AND
              the orientation is landscape AND
              devicePixelRatio is 1.5 or lower
              (two queries necessary) */
}
```

Width and Height

By far the most important media query you'll use is width. Use of height is typically restricted to specific use cases where something has to be shown on the site's home screen. The `width` and `height` media queries give the width and height of the current layout viewport and work in all browsers. After using the perfect meta viewport tag you can reliably pull out the width of the layout viewport, which is now equal to the ideal viewport. This is the core of responsive design.

```
@media all and (max-width: 400) {
       div.sidebar {
              // these div.sidebar styles
              // are used when the layout viewport
              // is 400px wide or narrower
       }
}
```

Height is more difficult to use, because it may take the browser toolbar into account, and that toolbar may slide into or out of the screen as the user scrolls. Feel free to use it, but give the browser a bit of leeway in determining the height.

Ems in Media Queries

The em unit in media queries deserves some special attention. At the time of writing it's very popular, but to my mind that popularity is somewhat overrated, and although there's nothing wrong with ems, they aren't inherently superior to pixels, either, except in one very specific use case.

In CSS 1em is exactly equal to the font size; for a 14px font, 1em will be 14px wide. The term font size usually equates to the font size of the element your CSS refers to, but in the context of media queries it means the root font size of the document; that is, the font size of the `html` element. Media queries are page-wide, after all, instead of applying to a specific element.

The default font size of the `html` element is 16px, so by default 1em is 16px wide. Of course, you can change the root font size: if you set it to 12px, 1em will be 12px wide; if you set it to 20px, 1em will be 20px wide, and so on.

On mobile, the root font size doesn't change when you zoom. Zooming is the process of enlarging CSS pixels, but that process has nothing to do with font sizes, so 1em will continue to be the same number of CSS pixels regardless of how much the user zooms. On mobile, then, there's nothing that makes ems inherently superior to pixels.

On desktop, it's more complicated. The page zoom we described above also increases the size of the CSS pixels without touching the font size, so here, too, ems are not superior to pixels. However, Firefox and Safari still support font size zooming, where only the font size is increased. So here is a genuine use case for ems over pixels. Another use case is the user setting a different, usually larger, font size in their browser preferences. Here, too, ems give you a better reading than pixels. Unfortunately, I do not know how many users on either mobile or desktop do so.

In other words, ems are superior to pixels only if the root font size of your site is likely to change and you want your layout to respond to that. If this is the case for your website, use ems. In all other cases it doesn't really matter if you use ems or pixels. Ems still work fine, and

to some they may be more logical than pixels when describing a layout, but they have no inherent superiority over pixels.

device-width and device-height

You should avoid `device-width` and `device-height` because they always, in all browsers, use the values of `screen.width/height`. As we'll see later, these JavaScript properties may give either the dimensions of the ideal viewport or the physical number of device pixels. Thus, it's impossible to predict whether you'll get the right or the wrong information. That makes `device-width` and `device-height` very dangerous to use.

device-pixel-ratio and resolution

We've already discussed resolution, and how it's the ratio of the ideal viewport to the screen size in device pixels. Thus the `resolution` media queries are useless for determining anything about the physical size of the device, although they can be used to determine if you want to send high-res images because the user is on a Retina-like screen.

There's a browser compatibility problem here: the WebKit-based browsers need `-webkit-device-pixel-ratio`, while all other browsers need `resolution`. Although `resolution` will win out in the long term, for the moment you still need `-webkit-device-pixel-ratio` as well.

> In Fall 2013 I conducted a viewport survey among my readers, and one of the questions I asked was which resolutions they checked for in their media queries. More than half of the respondents checked for a device pixel ratio of 1.5. So this seems to be some sort of emerging industry standard. See the survey results for yourself at http://smashed.by/mwhb7.

There's an additional trick: the unit. `-webkit-device-pixel-ratio` does not expect a unit; it's just an integer that corresponds to `window.devicePixelRatio`. On the other hand, `resolution` accepts the dpi and dppx units. We've already discussed those: 1dppx is equal to a device pixel ratio of 1, while 96dpi is equal to 1dppx. DPI is fully supported by all browsers while dppx is not, so it's best to use dpi. This is a cross-browser resolution check:

```
@media all and ((-webkit-min-device-pixel-ratio: 1.5),
       (min-resolution: 144dpi)) {
       // styles for when resolution is 1.5 or higher
}
```

Again, use a `min-` prefix here. You want to know if the resolution is 1.5 or higher, and not if it's exactly 1.5.

orientation

Let's do an easy case: `orientation`. This media query is meant for detecting the current device orientation, and it recognizes the keywords `portrait` and `landscape`. All browsers support it. No prefixes, no complicated stuff. So use it if you need it.

aspect-ratio and device-aspect-ratio

`aspect-ratio` and `device-aspect-ratio` give the aspect ratios of the layout viewport and `screen.width/height`, respectively. These ratios are expressed as a fraction; for example, 3/4 or 16/9. Be aware that incoming or exiting browser toolbars may change the layout viewport's aspect ratio.

JavaScript

To close this long and complex chapter we need to talk about JavaScript properties. Just about everything we covered in this chapter can be read and used by JavaScript, provided you know the correct properties to query.

I tested media queries extensively, and it turns out that all of them are slaves to certain JavaScript properties in just about all browsers. Also, you need to know about the `orientationchange` and `resize` events.

Once upon a time, when the height of fashion among web developers was bearskin and our WYSIWIG editor of choice a clay tablet, Netscape and IE fought for dominance on the browser market. They used proprietary extensions to force web developers into their camps, so that many interesting JavaScript properties actually had two forms: one for Netscape and one for IE.

This also happened to the viewport properties. Netscape insisted that it was `window.innerWidth`, while IE maintained it was `document.documentElement.clientWidth`. JavaScripters had to use both if they wanted to read out the browser window size. Finding the screen size was easier: both browsers used `screen.width`.

Time went on, the importance of standardization was discovered, and in the spirit of detente the browsers started to support each other's properties. Nowadays, all desktop browsers support both

`window.innerWidth` and `document.documentElement.clientWidth`, and the only difference is that the first property includes the scrollbar width while the second excludes it.

While this was a nasty situation at the time, it's actually very useful to have several properties now that we have several viewports to measure. We can use one property pair for each of the viewports and expose all of them to curious web developers. Thus we arrive at today's system:

1. `document.documentElement.clientWidth/Height` returns the dimensions of the layout viewport. Universally supported.
2. `window.innerWidth/Height` returns the dimensions of the visual viewport. Near-universally supported.
3. `screen.width/height` returns the dimensions of the ideal viewport. Serious browser compatibility problems.

The Layout Viewport

`document.documentElement.clientWidth/Height` is universally supported and gives you the dimensions of the layout viewport. That can be very useful in cases where you want to use JavaScript, and not a media query, for your page logic. For instance, the following is the JavaScript equivalent of `@media all and (min-width: 600)`:

```
if (document.documentElement.clientWidth >= 600) {
      // load Twitter and Facebook widgets
}
```

This is sometimes useful, as JavaScript is a much better tool for some jobs than media queries. For example, if you want to download and show a third-party widget only if the layout viewport is wide enough, you should use JavaScript. In CSS, the best you can do is set `display: none`, but many browsers will still load the assets and take up valuable bandwidth, even though they're not used in the page. In JavaScript, you can postpone the download until you're certain the layout viewport is in fact wide enough to display the widget properly.

The Visual Viewport

`window.innerWidth/Height` is well-supported, but not universally. The most serious problems are with Android WebKit 2 and the proxy browsers. Fortunately, finding the visual viewport dimensions is something you don't want to do too often. Right now, I think it's only useful when you have a complicated layout where the zoom level matters a lot — very rarely, in other words. But if you need it, it's there.

The Ideal Viewport — or the Screen Size

And then we get to the real problem: `screen.width/height`. This can mean two things, depending on the browser:

1. The dimensions of the ideal viewport.
2. The screen size in device pixels.

In practice this means that you can't use `screen.width/height` at all, since you never know what you're going to get. Even worse, it affects analytics tools, too. Such tools commonly read out `screen.width/height` to give you an overview of the screen resolutions users of your site have. Unfortunately, you'll sometimes get the

ideal viewport instead of the physical resolution you'd expect. Thus, resolution statistics in analytics tools are completely unreliable and should be ignored.

Although I think the first definition, dimensions of the ideal viewport, is going to win out in the future, that hasn't actually happened yet — and Android WebKit, not being maintained any more, will never use the ideal viewport definition. So using `screen.width/height` is not really possible until Android WebKit has died out.

Remember that the `device-width` and `device-height` media queries use the values `screen.width/height` provide, regardless of the definition the browser uses. All the problems sketched above also go for these media queries.

devicePixelRatio

We've already encountered `window.devicePixelRatio`: its unitless value gives the ratio between the physical screen size in pixels and the ideal viewport. The `device-pixel-ratio` media query uses the same value, as does the `resolution` media query if you use the dppx unit.

Changing the Meta Viewport Tag

In most browsers it's possible to change the contents of the meta viewport tag. Assuming the meta viewport is the first meta tag in your document, do this:

```
var meta = document.getElementsByTagName('meta')[0];
meta.setAttribute('content','width=400');
```

Now the layout viewport width is set to 400 in most browsers. At the time of writing, IE and Firefox don't support this, and neither do old BlackBerrys, but otherwise support is widespread. Note that it is not possible to remove the meta tag entirely so that the layout viewport returns to its default width. However, you could set it to a fixed value of, for instance, 980px or 1,024px if you want to offer "go to desktop layout" functionality.

The orientationchange Event

The `orientationchange` event fires whenever the user changes the device orientation — in all WebKit- and Blink-based browsers. At the time of writing, neither IE11 nor Firefox 31 support it.

The resize Event

The `resize` event fires whenever the viewport is resized. But which viewport? Not surprisingly, browsers disagree on that.

> True browser compatibility junkies will appreciate the following conundrum: if I rotate the device 180 degrees, should the `orientationchange` event fire or not? On the one hand, the device's orientation changes. On the other hand, the final orientation is the same as the starting orientation: you go from portrait to portrait or landscape to landscape. Browsers come to different conclusions. What do you think?

The ideal viewport cannot be resized: it is what it is. That leaves the layout and visual viewports. It turns out that most browsers fire the `resize` event when the layout viewport is resized, but not when the visual viewport is resized (that is, the user zooms). This rule is not absolute, and to make things more complicated the several methods of resizing the layout or visual viewport may not all fire a `resize` event. To make things even more complicated, Safari fires a `resize` event when the `html` element, which is not a viewport, is resized by adding or removing content. This is definitely something that shouldn't happen.

You can find the gory support details on my site. For now, we can say that it's best to distrust the `resize` event on mobile. It is too erratic across browsers to be of much use.

***

With the viewports explained we can now turn our attention to a few CSS declarations that are different on mobile and desktop.

Concept	Description	JavaScript property
Physical screen	The dimensions of the device screen in device pixels.	Officially none; sometimes `screen.width` and `screen.height`
Layout viewport	The size of CSS's initial containing block. All percentage CSS widths are derived from it.	`document.documentElement.clientWidth` and `-Height`
Visual viewport	The current size of the part of the page the user sees on the screen. Influenced by zooming.	`window.innerWidth` and `-Height`
Ideal viewport	The dimensions the layout viewport should get in order to deliver a perfect user experience for the device.	`screen.width` and `screen.height`, but not in all browsers
Resolution	The ratio between the physical screen size and the ideal viewport size.	`window.devicePixelRatio`
Orientation	The current device orientation: portrait or landscape.	`window.orientation`
Zooming	The zoom factor of the page, relative to the ideal viewport.	Sometimes `screen.width / window.innerWidth`

Media query	Meta viewport	Browser compatibility
Officially none; sometimes `device-width` and `device-height`	-	See Ideal viewport.
`width` and `height`	`width` directive sets its width	all
-	-	most
`device-width` and `device-height`, but not in all browsers	`width=device-width` sets layout viewport to ideal viewport	In some browsers, JavaScript and media queries use the ideal viewport dimensions; in others, the physical size of the screen in device pixels.
`-webkit-device-pixel-ratio` and `resolution`	-	WebKit-based browsers need the first media query; the others the second. Eventually only resolution will remain.
orientation	-	Media query supported by all. JavaScript property supported by most.
-	`initial-scale`, `minimum-scale`, `maximum-scale`	JS very unreliable. Meta viewport directives decently supported, but issues in Android WebKit and IE.

Chapter 5

CSS

Chapter 5

CSS

Just like their desktop cousins, mobile browsers support CSS. Their support doesn't differ all that much from the desktop: all rendering engines, and thus all browsers, support margins, colors, font sizes, floats, and all other stock CSS declarations. You won't have any problems with declarations like that (except for the ones caused by the mobile screen being much narrower than the desktop screen, but those are not technical differences).

Still, there are a few cases where CSS support on mobile browsers is different by necessity, and in this short chapter we'll take a look at a few of them. Although these declarations are interesting in themselves, and studying them will teach you important lessons about mobile web development, the main point I'm trying to get across is the kind of thinking mobile browser vendors must engage in before supporting them. Their decisions influence the way you can use CSS in their browsers, so it's important to understand their point of view.

There are four main reasons why support for CSS declarations may differ from desktop to mobile:

1. The use case they serve does not exist on touchscreens, or does not fit there very well. Example: `:hover`.
2. The viewport is involved, but the specification fails to mention which viewport. Example: the `vw/vh` units.
3. They require an independently scrollable layer, which is much harder to achieve within the constrained resources of a mobile phone than on a desktop computer. Example: `background-attachment`.
4. Hardware constraints, especially when it comes to memory and GPU. Transitions and animations, notably, may fail in such environments. This is a device problem, and not a browser problem, and is therefore different from the rest.

The CSS specifications are not always very useful in these situations. Most of them handle touchscreen or mobile use cases badly because they were written in the context of mouse, keyboard, and traditional display. Sometimes that context doesn't matter, but at other times it matters a lot. We'll encounter a few problems below.

Finally, a note on browser compatibility. As usual you can find the compatibility information you need on this book's companion site at http://quirksmode.org/mobilewebhandbook. In the rest of this chapter I'm quite vague about compatibility details because they'll likely change between me writing this book and you reading it. If you need exact information, look it up on the site.

position: fixed

Let's start with `position: fixed`, a declaration many web developers would love to use on their mobile sites but can't — and maybe shouldn't. The W3C has the following to say on the subject in the CSS 2 specification, first published in 1998, well before the mobile web amounted to anything:

> *The box's position is calculated according to the 'absolute' model, but in addition, the box is fixed with respect to some reference. [...] The box is fixed with respect to the viewport and does not move when scrolled.*

Since `position: fixed` support differs so much among mobile browsers, and illustrations don't help much in conveying what actually happens, I prepared a few videos on the companion site. These will show you what's going on; something that no amount of words can do.

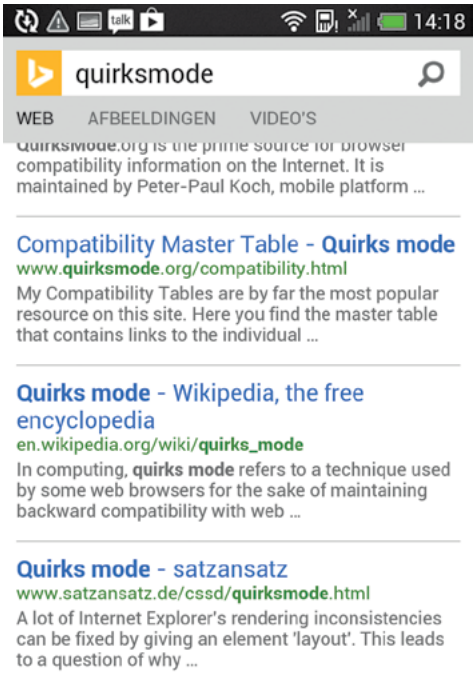
Which viewport is meant here? The spec doesn't say so explicitly, but a fixed layer is expected to stay in sight, so it's logical that the visual viewport is meant here. Of course, the fact that it's logical doesn't necessarily mean it's implemented everywhere.

Although mobile browsers already recognized `position: fixed` when I started my mobile research in 2009, their support was weird and buggy. In particular, many browsers positioned fixed elements relative to the layout viewport, and not the visual one. It took until 2013 for the first perfect implementation to be released in Chrome and Opera.

Positioning a fixed layer relative to the layout viewport may seem like a rather gross error that even a cursory study of the spec should have

prevented, but the problem is something else entirely. Although the fixed layer doesn't scroll when the user swipes, the rest of the page does. So in order to support true fixed positioning, browsers must be able to scroll layers independently of each other. Back then browsers couldn't do that, so they had to choose between a bad implementation of `position: fixed` or none at all. They chose the bad one.

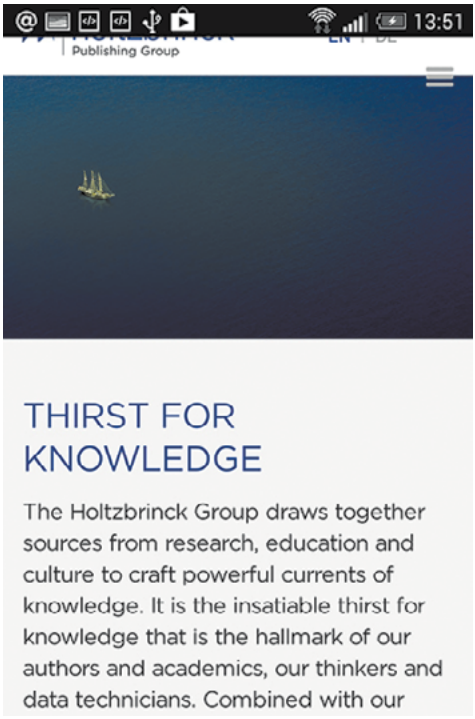
So old browsers get it wrong, and some new ones get it right. Unfortunately that's not all: there are a lot of implementations spanning the entire spectrum from mostly wrong to mostly right, while Safari has its own unique take that doesn't resemble anyone else's and doesn't make a lot of sense. See the companion site and the videos for a complete overview.



Bing.com uses a fixed bar at the top of the page that doesn't interfere with the rest of the page.

And now for the real question: what happens when the user zooms? The spec is silent on this, but since the layer is fixed relative to the visual viewport it makes sense to let it scale with it so that, for example, one with `width: 50%` continues to cover one half of the visual viewport regardless of zoom level. This may be what you want, but it could also be surprising.

So what is a poor web developer to do with `position: fixed`? A tasteful top bar of 100% width and not too much height, containing two or three links or other items seems the most you can hope for. The fact that it's small means it won't take up too much valuable screen real estate, and the fact that its coordinates are 0,0 solves a lot of compatibility problems.



The Holtzbrinck site restricts itself to the "hamburger menu" at top-right. Here, as with Bing.com, the point is to keep one or two vital links in view all the time, while taking up as little space as possible. This use of `position: fixed` is somewhat different from the desktop, where a fixed layer usually contains a lot of extra features. Such layers don't work on mobile because of the small screen, so restricting yourself to important features works best.

Still, in my opinion, its use is somewhat overrated. The problem with fixed elements is that they take up so much space on the small screen. Although it could make sense to keep your branding in view all the time, it's only really possible with simple sites. So be careful how you use it.

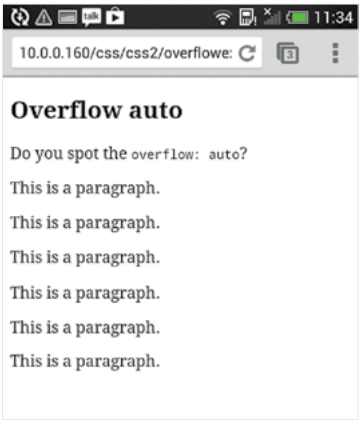
For a long time I have wondered whether mobile is different enough from desktop to warrant a different `fixed` implementation; call it `position: device-fixed` (see http://smashed.by/mwhb13 for my article). It would work like `fixed` does in modern browsers, but it would not scale the font (and possibly other elements such as images). Thus, the `device-fixed` layer would essentially stay the same, no matter how the user zooms.

To my surprise, `device-fixed` was implemented in IE11. Although at the time of writing it suffers from lag and isn't quite ready for prime time yet, the Microsoft implementation at least allows you to figure out if this is a feature that's useful enough to retain.

overflow: auto

The case of `overflow: auto` is the clearest example of layers that have to scroll separately from the rest of the page. If an element has this style it must be scrollable separately from the rest of the page. If a browser does not allow scrolling, users cannot access part of the content, and that's of course very bad.

Once upon a time many mobile browsers did this wrong. They cut off the content at the right spot, but did not allow independent scrolling and thus made parts of pages inaccessible. Meanwhile things are getting better: just about all mobile browsers support `overflow: auto` correctly. The most important exceptions are the proxy browsers — their clients will never support independent scrolling because they show only a single image.

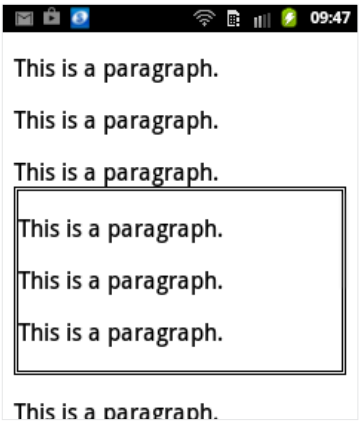


This page contains an `overflow: auto`, and one of the paragraphs is currently hidden. There's no way of knowing that as a user, unless you happen to scroll in the right place.

A border could help to distinguish the element, but if the browser doesn't support independent scrolling of the layer, the user still cannot see the hidden paragraph.

The old, Presto-based Opera Mobile found a solution that does not require scrolling: if `overflow: auto` is defined, just stretch up the element until all the content fits. This is not beautiful — in fact, it might break some layouts — but it's an accessible one. I wonder why the proxy browsers, Opera Mini in particular, never implemented this.

All in all `overflow: auto` is not very suited to mobile interfaces, and even though it might technically work, it's best to quietly forget about it.



Opera Classic, which doesn't support independent scrolling of the layer, stretches up the element so that the content is visible. Although this might break some layouts, it's the least bad solution when it comes to accessibility.

overflow-scrolling

Although most browsers have a pretty smooth, kinetic scroll that you can use straight away, Safari and a few other browsers do not. Their scrolling is stilted and non-kinetic. This is caused by processor constraints: it's more computationally expensive to do a kinetic scroll, and disabling it by default helps save battery life.

Still, users and web developers want a nice scroll effect. That's why Apple created (and BlackBerry copied) the `-webkit-overflow-scrolling: auto` CSS declaration that adds kinetic scrolling to an element. Thus, expensive kinetic scrolling is only enabled when the web developer explicitly asks for it.

I wonder if this declaration will make a lot of difference in the long term, since web developers will start to apply this declaration to all their scrolling layers, and we'll end up with the same result as when browsers support kinetic scrolling by default.

background-attachment

A similar problem exists for `background-attachment`. There are three values, two of which create a separately scrollable background image:

1. `scroll`: the default. The background image scrolls with the page.
2. `fixed`: the background image is fixed relative to the viewport, so that the element serves as a kind of window on the image, and scrolling effectively pans that window so that you see a different part of the image. Question: which viewport?
3. `local`: the background image scrolls with the element.

It's clear that any `local` background image is an independently scrolling layer since it scrolls with the element, and not with the page. But what about `fixed`? Which viewport is it relative to? If it's the layout viewport, it's indistinguishable from `scroll`, so it's clear that it should be fixed relative to the visual viewport. That, however, again creates an independently scrolling layer.

The problem once more is that mobile browsers cannot support too many independently scrolling layers. Both `fixed` and `local` could create many of them on one page, which is why most browsers support only one of the two. In my tests I found only a single browser, UC, that supports both, and a few browsers that support neither.

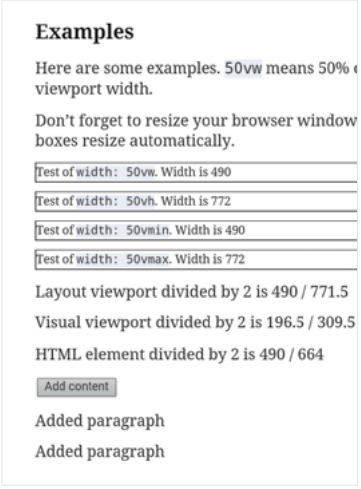
Again, illustrations don't help much in conveying what actually happens with `background-attachment`. So I added a few more videos on the companion site to show you what's going on.

In other words, `background-attachment` is unreliable on mobile and I advise you not to use it — or to make sure that the effect is nice to have instead of required. Besides, even if all browsers implemented it, background images in desktop-first websites could start to overlap due to the lack of space on the mobile screen. So that's an extra reason for not using this declaration.

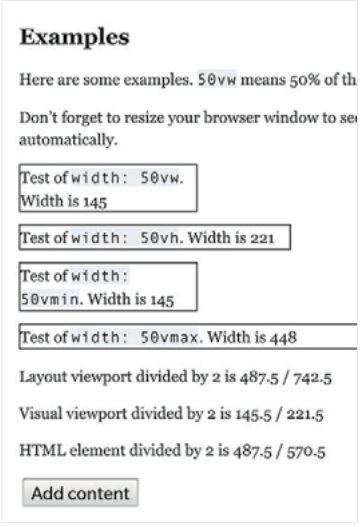
vw and vh

The vw and vh units denote percentages of the viewport. Thus, `50vw` means 50% of the viewport width; and `20vh` means 20% of the viewport height. But which viewport? All things considered, it should mean the layout viewport. If it were the visual one, the widths and heights of elements would change every time the user zooms, and apart from being computationally very expensive it would not make sense at all to the user.

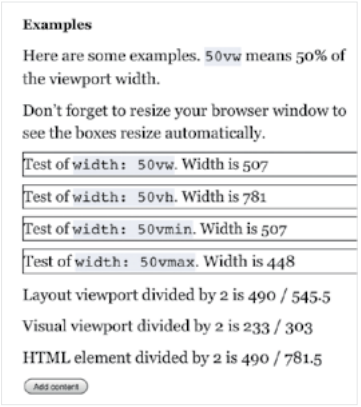
Not many mobile browsers support these units as yet. At the time of writing only the Blink-based browsers, IE, and Firefox on Android support them correctly. A few browsers support them relative to the visual viewport, which gives a weird effect, and Safari does its own thing by sometimes,



Chrome gets it right: the units are resolved relative to the layout viewport, and thus don't change.



BlackBerry gets it wrong: the units are resolved relative to the visual viewport, and thus change when the user zooms. This is cool, but likely not what the user wants.



Safari gets it weird: the units are resolved relative to the `html` element. If that element grows because content is added to the page, the units change.

but not always, making them relative to the size of the `html` element, which may change when more elements are added to the page. (This behaviour is rumored to be removed from iOS 8, but I haven't tested that myself.)

So vw and vh are not really ready for mobile use. That's a pity, since they could be very useful units for responsive designs. (In fact, I suspect the units were invented specifically for this purpose.)

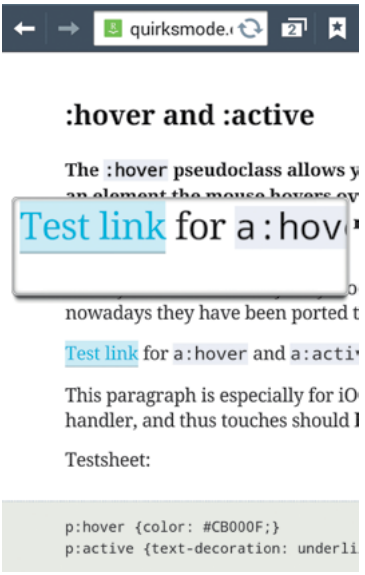
:active and :hover

Finally we have to discuss `:active` and especially `:hover` — desktop concepts that translate fairly badly to touchscreens, but that are usually safe to use because the worst that can happen is that they don't work. Unintended side effects do not occur.

What does `:hover` mean on a touchscreen? Your finger is either on the glass or it's not; it's either tapping (clicking) an element, or it's completely absent. Technically, it's possible to detect the coordinates of a finger hovering above the device, but few devices do so, and the ones that do don't seem to share this information with other browsers. And even if all devices handled this properly, your finger would still be over the hover effect while it's taking place, so you wouldn't be able to see it.

Still, `:hover` is so widely used that mobile browsers felt forced to implement it, kind of. That's why `:hover` styles are added to an element when the user touches it, at the end of the event cascade that we'll encounter in the Touch And Pointer Events chapter. When the user subsequently touches another element, the `:hover` styles are removed from the original element. (The `mouseover` and `mouseout` events are treated in the same way — rightly so, since they're the JavaScript equivalents of `:hover`.)

As to `:active`, on desktop it applies to elements that the user is currently clicking on (or focusing on with the keyboard, but that's a more complicated use case). A one-to-one translation from desktop to mobile would be applying the styles when the user touches an element, and removing them when that touch stops. This is exactly what the supporting mobile browsers do, although the effect will usually be hidden by the user's fat finger. Many browsers don't support it at all, however, and I find it hard to argue with them.



On its latest Galaxy models (S4 pictured here), Samsung implemented its own hover effect, where parts of the page are enlarged, and the details are shown well above your fat finger. (Turn it on in Settings → My Device → Air View.) That's cool, and it might even be useful, but it does not trigger CSS `:hover` effects, and only works in the default browser.

So `:hover` doesn't really work on mobile — and certainly not like you're used to on desktop. Whether it's safe to use depends on what you're doing. If you're just changing a link style there's no problem: it will occur, though maybe not at the time you expect. More problematic are extra bits of content that pop up when the user hovers. This effect may not work on mobile, and even if it does it's unintuitive, and the content may be hidden under the user's finger. You should find another solution for showing extra content.

Another safe technique is the combination of `:active` and `:focus`, which on desktop is commonly used to create a kind of hover-with-keyboard effect. You can use this on mobile. `:focus` works fine since all browsers support it, except the proxy browsers. Proxy browsers would have to go back to the server to change to `:focus` styles, and the creators decided the effect is too small to warrant a round trip.

Transitions And Animations

Finally, a quick word about transitions and animations. The problem with them is not the browser but the devices. Browsers support them fine, but in order to create a truly smooth effect they have to be able to connect to the device's GPU. On modern high-end smartphones this is no problem, but some older or cheaper phones may not have the required hardware or system APIs, leading to stilted effects.

Since this depends on the device, it's not a problem that can be captured in a browser compatibility table. One browser on a high-end device may support transitions and animations perfectly, only to fail when it's installed on a low-end device.

As a general rule, test on your oldest and crappiest device whenever you use transitions and animations. Besides, even modern devices may falter when confronted with too many animations on one page. Although they allow access to the GPU, their hardware is still slower than desktop hardware. My general advice is not to use animation-heavy pages on mobile; if you absolutely have to, test them on a wide range of devices.

***

We have now dealt with some of the most important unique CSS features of mobile browsers, and it's time to turn to JavaScript and the touch and pointer events.

Chapter 6

Touch And Pointer Events

Chapter 6

Touch And Pointer Events

When Apple released the first true touchscreen browser back in 2007, it also delivered **touch events** to monitor the user's touch actions.

There are W3C recommendations for both touch and pointer events, so both are web standards. They can be found at http://smashed.by/mwhb8 and http://smashed.by/mwhb9, respectively.

However, it seems the W3C is transitioning away from touch events and towards pointer events. The Web Events Working Group that produced the touch events specification has been disbanded and work on the spec had ceased, while the Pointer Events WG is still a going concern. Pointer events are becoming the standard as far as the W3C is concerned.

Google and Mozilla are working on an implementation of pointer events — maybe they're already finished by the time you read this. See the Chrome discussion at http://smashed.by/mwhb10 and the Firefox discussion at http://smashed.by/mwhb11

Most other browser vendors copied them, except for Microsoft, which invented **pointer events**. These two sets of events are the topic of this chapter.

Although at first sight the pointer events may seem yet another instance of IE-is-different-just-because, that's not the case. Microsoft is making an interesting philosophical point here that we'll discuss at length. At the time of writing Google and Mozilla are considering implementing pointer events, and the W3C is transitioning from touch to pointer events as well.

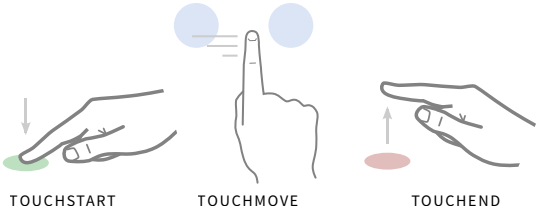
In most respects, touch and pointer events are normal JavaScript events. They fire when a touch action occurs, you can assign event handlers to them, and their event objects give useful information about the touches. There are a few technical differences between touch and traditional mouse or keyboard events. Also, for reasons of backward compatibility, touchscreen devices must fire mouse events because so many websites depend on them. But when do you fire mouse events on devices that don't have a mouse? Part of this chapter is devoted to discussing these issues.

The rest of the chapter is more philosophical in nature. Along with the iPhone, Apple introduced a new interaction mode, touch, which now coexists with the traditional mouse and keyboard interaction modes. Web developers must make sure their sites work with all three. At first sight, touch events seem to be roughly the same as mouse events. What are the differences? Do we need separate events for separate interaction modes?

Touch Events

Let's start with touch events, since they are better supported than pointer events. Later, we'll see that the pointer events are pretty similar. There are four touch events:

1. `touchstart`, which fires the instant the user's finger touches the screen.
2. `touchmove`, which fires continually while the user is moving their finger.
3. `touchend`, which fires the instant the user's finger releases the screen.
4. `touchcancel`, whose meaning depends on browser. Discussed below.



`Touchstart`, `touchmove`, and `touchend`. The `pointerdown`, `pointermove`, and `pointerup` events fire at exactly the same time.

These events are supported by most touchscreen browsers, with the main exception being IE. A few very old or bad browsers, such as Symbian Anna's default browser, don't support them. The proxy browsers don't support them either because these events don't fit the proxy browsing model. We discussed the reasons in the Browsers chapter.

touchcancel

I admit I do not understand the `touchcancel` event. It fires when a touch sequence is canceled, but what that means is very much up to the individual browser. For instance, Chrome fires it when the user's touch leaves the screen, but most other browsers don't.

Fortunately, I have never found a good reason to use this event, and it seems scripts and libraries hardly use `touchcancel`, either — the ones that do treat it as an equivalent of `touchend` and include it just to be on the safe side. Therefore, this chapter will ignore the `touchcancel` event. If you ever run into weird problems because browsers don't see `touchend` events in certain situations, you can always bind your `touchend` event handler to the `touchcancel` event as well.

Gesture Events

In addition to touch events, Safari on iOS also implements the `gesturestart`, `gesturechange`, and `gestureend` events, while IE has a slew of similar events. A gesture event is defined as two or more touch events taking place simultaneously.

There are two problems with these events: no other browser supports them, and they are rather useless. In theory it sounds great to detect user gestures, but in practice you have to figure out what the user is trying to achieve by studying the touch coordinates and how they change over time. We don't need the gesture events for that: ordinary touch events give us the same information. For these reasons the gesture events are not important and this chapter will not cover them.

Other Events

At one point, the touch event specification contained the `touchenter` and `touchleave` events, which would fire if the user's touch entered or left a certain element. They have never been implemented, although IE supports the Microsoft alternatives. Since these events are a good idea, I hope they'll make a comeback. In certain interfaces it would be useful to know if a user's finger slides into or out of an element.

What we could do with is a `zoom` event that fires when users zoom (or rather, when they stop zooming). I've been been saying this since 2010, but so far nobody has listened. Still, it would be good to know if the user zooms — perhaps you want to change the interface a bit, or you just want to collect zoom data in order to find out if your font is too small.

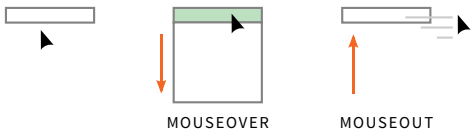
Example Scripts

We're going to use three example scripts to show how touch and pointer events compare to mouse and keyboard events. Studying them will also teach you to think about interaction modes and how to port mouse-based effects to touch and vice versa.

Drop-down Menu

The first script is a golden oldie: a drop-down menu. Like it or loathe it, it's ubiquitous on the web, and it's also the perfect example script because it encompasses so many crucial aspects of event handling.

Traditionally, a drop-down menu works with the mouse. The user hovers the mouse pointer over an item and a menu folds out. The user removes the mouse pointer, and the menu folds in. It should also work with the keyboard: the user tabs to the item with the keyboard, and as soon as it has focus (the focus event), the menu folds out; if focus is re-



The traditional drop-down menu opens `onmouseover` and closes `onmouseout`.

moved (the blur event), the menu folds in. This is harder to implement than the mouse effect, but it's possible.

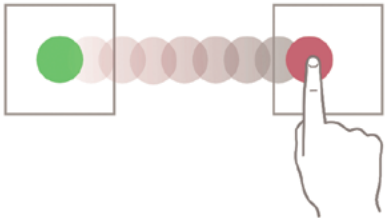
But how do we port a drop-down menu to touch-based interaction? Substituting `touchstart` for `mouseover` and `touchend` for `mouseout` does not work. Touching an item folds out the menu; no problems there. But once the menu is open, users want to touch a specific link. They lift their fingers and the menu folds in. That doesn't work. And if we left the menu open `ontouchend`, when would it close?

The best solution for a cross-device environment is to work with the click event. Click, and not `mouseover`, would open a menu, and clicking on another menu item would close it again. As we'll see later in this chapter, the `click` event is perfectly safe, and as an additional bonus the drop-down menu will work roughly the same with mouse and touch.

Still, mobile browsers will have to contend with the fact that there are tens of thousands of mouseover-based drop-down menus on the web. Fortunately, drop-downs were a specific use case Apple had in mind when designing the touch event cascade, and all other browsers copied its solution, which we'll encounter later in this chapter.

Drag and Drop

Just about the polar opposite of a drop-down menu is a drag and drop script. The user takes an element (`mousedown`), moves it elsewhere (`mousemove`), and drops it (`mouseup`), after which the script calculates if the current spot is a valid drop target and does something based on the outcome.



Drag and drop is pretty intuitive with the mouse, but even more intuitive with touch.

Porting this to touch is very simple: just make sure that `mousedown` is paired with `touchstart`, `mousemove` with `touchmove`, and `mouseup` with `touchend`. This works fine, and the only minor problem is finding the event coordinates. We'll get back to that.

The problem here is keyboard accessibility. How do you allow keyboard users to move the draggable elements? You could make one area of an element keyboard-focusable, and once the user focuses on it you can start listening to the arrow keys. Technically this is not very difficult, but the user experience goes down in flames. This problem is essentially unsolvable: the drag-and-drop metaphor is based on mouse or touch interaction, and just doesn't work on keyboard.

In case you're curious, here's a script that implements drag and drop for mouse and keyboard. I wrote it a long time ago: http://smashed.by/mwhb14. The keyboard drag and drop is not really intuitive, but I haven't found a better solution yet. Adding the touch events to this script is very simple, and is left as an exercise for you.

Scrolling Layer

In 2011 I needed a horizontally scrolling element that worked on all devices. Back then, this required a script, so I wrote one. Meanwhile, native scrolling has improved such that a script is no longer necessary; we saw that in the CSS chapter. Still, a scroll script is a useful example, so we'll pretend we still need it.

The speaker photo bar can scroll horizontally.

Writing the scroll script itself was not a problem. `Ontouchstart`, calculate the current position of the scrolling element and initialize the other event handlers; `ontouchmove`, scroll the element the same number of pixels as moved by the touch; `ontouchend`, run a special function that calculates a pleasant deceleration, and once the element comes to a stop the function ends. Easy. Took me about two hours.

But what about non-touch devices? To get it working I had to translate the interaction from touch to mouse and keyboard. Keyboard was easy. I just listened to `keydown` events, and scrolled the element when the user used the left or right arrow keys. This may not be easily discoverable (I hate explanations of this sort of functionality and never include them), but formally the script is now keyboard-accessible.

But what about the mouse? Technically, it's trivial to add the `mousedown`, `mousemove`, and `mouseup` events to the script, but the interaction would be very odd. The user would have to move over the element with the mouse button depressed in order to scroll. This is the same

interaction as for drag and drop, but in the case of a scrolling layer it's not intuitive.

I could use the old-fashioned arrows to the left and right of the scrolling element, where mousing over the arrows would engage the scroll script, but that would be visual clutter. Besides, I'd really have to hide the arrows when the user used touch or keyboard, but I couldn't figure out how to do that safely and correctly. (We'll get back to this problem later.) In the end, I decided not to create a mouse interaction at all.

Events And Interaction Modes

Back in 1996 Netscape introduced mouse events and the famous mouseover effect, and web developers saw that it was good. Then accessibility specialists spoke up, pointing out that some people do not use a mouse, and that browsers also had keyboard events. Some web developers complied, and from that moment on they coded for two interaction modes: mouse and keyboard. Then Apple added the touch interaction mode, bringing the total number of interaction modes to three.

Web developers must make sure that their sites work with all three interaction modes. Sometimes it's easy, sometimes it's hard, but it's always necessary — not only for your current websites, but also to start thinking about the translation of a UI element to the various modes. I hope the notes I gave for the example scripts show you how to think about these issues.

There's no reason why we couldn't have many more interaction modes in the future. Take the Xbox Kinect, which translates body movements to screen actions, so that you can use your hand to steer a cursor on the

screen. Technically, steering a cursor means using mouse events, but from a user's perspective it might count as a new interaction mode. It feels different, after all.

Cars, fridges, wearables, and any other kind of emerging device may bring new interaction modes to users and web developers. `doorclose` event, anyone? (In fact, inventing new JavaScript events is a fun game for post-conference parties.)

Thinking about interaction modes and JavaScript events leads to three questions:

1. Does every interaction mode need its own events?
2. Will devices support JavaScript events for legacy interaction modes even if they don't make sense on the device?
3. How do you find out which interaction mode(s) the device supports, or the user is currently using?

Right now, the answers are: yes, yes, and it's complicated. Still, the first answer might become no in the near future. Look again at the Kinect: will we have entirely new handwave events, or will we use pointer and mouse events? Technically, a cursor is a cursor, no matter how the user moves it.

Event Equivalents

Right now, each interaction mode has its own set of events. Still, that does not mean they are totally and irreconcilably different. In fact, there are equivalences between certain events. The table gives a general overview.

Mouse	Touch	Keyboard
mousedown	touchstart	keydown
mousemove	touchmove	keydown/keypress
mouseup	touchend	keyup
mouseover	-	focus
mouseout	-	blur

Event equivalents

It's clear that the touch sequence `touchstart–touchmove–touchend` resembles the mouse sequence `mousedown–mousemove–mouseup` and (up to a point) `keydown–keypress–keyup`, and that's not coincidental. All three interactions can be described as start–move–stop, and thus the event sequences are pretty similar. (Then we don't need different events, right?)

Still, sometimes two of the modes resemble each other, but not the third. In a drop-down menu, mouse and keyboard are similar, while touch is different. In a drag and drop script, mouse and touch are nearly the same, but keyboard is very different. And the three don't resemble each other much in the scrolling layer example. (So we need different events after all, right?)

Finally, there's the problem of `mouseover` and `mouseout`. `Focus` and `blur` are their keyboard equivalents, more or less, but there is no touch equivalent. In fact, as we saw in the CSS chapter, the concept of hovering does not exist on touchscreen devices.

A Touch of Difference

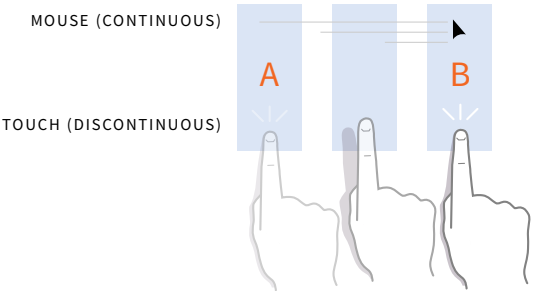
So event equivalence sometimes exists, depending on the context, but touch, key, and mouse events are not the same. Since keyboard is clearly the most distinct of the three, and web developers tend to concentrate on mouse and touch, let's discuss the differences between those two.

When the mouse pointer moves into an element, or the user clicks a mouse button, it's immediately clear what's going on and which events should fire. Not so with touch actions: they are overloaded with meaning. At the instant your finger touches the screen, the OS and the browser have no idea what's going to happen next. Do you want to tap the element? Or start a scroll, or a pinch-zoom action? Or do you want to double-tap? The browser must wait a little while before assigning meaning to your touch, and that interval is noticeable. We will get back to this — oh boy, will we!

Several touch actions may take place simultaneously. This is impossible with mouse actions: a computer has but a single mouse. Usually, this does not matter much: most sites only support single-touch interactions, which are easy to emulate with a mouse. Even if you have two sliders on one page, they do not interfere with each other, and if the user slides both at the same time you can treat each as an individual system, and both systems work equally well with touch or mouse.

It's different when a site allows, or requires, multi-touch interactions. If a script translates several touches taking place at the same time to a gesture, such as rotate or pinch-zoom, these effects cannot be replicated with a mouse. How much of a problem that is depends on the site and the use case, but it's important to be aware of it.

A touch event is coarser than a mouse event. The mouse cursor always points to exactly one pixel, while your touch may overlap several of them. Usually, the OS takes the central pixel as the one you touched for calculating coordinates and such, and gives you a bit of leeway for moving your finger between `touchstart` and `touchend` — but coming up is the horror story of a browser that didn't do this.



A mouse cursor moving from A to B will pass through the central element. A touch will not; the user has no need to touch the central element.

Touch events are discontinuous while mouse events are continuous. When you move the mouse cursor from element A to element B you have no choice but to move over all elements in between. The mouse movement is continuous and can be tracked by a script. When you move your touch from A to B, however, you usually release element A and touch element B without handling any elements in between: touch movement is discontinuous. This is, in fact, the problem we face while porting drop-down menus to touch events. Drop-downs expect continuous events because they were conceived for a mouse-only environment.

Touch events could carry a lot more information than mouse events. For instance, a touchscreen device browser may give information about your finger's temperature, the radius of your touch, and the pressure you exert. They don't actually do so now, but that could change in the future — especially since some property stubs are available in IE's pointer events.

In any case, I hope I convinced you that mouse and touch, while similar, are not the same.

Merging Touch and Mouse?

We have found that mouse and touch events are usually quite similar, but that there are small and sometimes significant differences between them. We need this background in order to understand Microsoft's pointer events and the point of view that gave rise to them.

Microsoft's contention is that mouse and touch do not necessarily need separate events. Therefore, the pointer events fire whenever the user changes something with the pointer — and pointer can mean a mouse cursor, a touch, or even a pen (or stylus). So this is Microsoft's take on the event equivalents:

Mouse Touch	Keyboard
`pointerdown`	`keydown`
`pointermove`	`keydown/keypress`
`pointerup`	`keyup`
`pointerover`	`focus`
`pointerout`	`blur`

Event equivalents according to Microsoft

We now have two fundamentally different approaches: Apple's, where mouse and touch are separated; and Microsoft's, where mouse and touch are integrated. For now, only IE supports the Microsoft model, while all other browsers support the Apple model. As we saw, Mozilla

and Google are considering the implementation of pointer events, so the situation might change in the future.

The reason Google considers the pointer events is interesting. Like the Microsoft Surface, the Chromebook Pixel has a touchscreen but also a keyboard with trackpad. Thus, both devices allow you to use mouse and touch events in your interactions, and even switch between the two. Websites should keep track of both sets of events, and that's a lot easier when they are merged into a single set.

The Microsoft Surface is a touchscreen tablet, but you can attach a keyboard with a trackpad. You can switch between mouse and touch while using a website, and that's a use case addressed by pointer events.

Does Microsoft have a good idea here? Personally, I tend to think so. It's likely that as time progresses, more and more devices will have both mouse and touch interactivity, so pointer events are forward-thinking. Also, they could easily be extended to cover other interactions as well. Currently pointer events also work with a pen (or stylus), and not only when a pen touches the screen, but also when you work with Wacom tablets and such. In the future, they could easily include a moving TV remote or Kinect gestures that steer a cursor and activate elements such as links. (A closing door would fall outside their comfort zone, though.) So pointer events could prove to be more future-friendly than separate mouse and touch events.

Let's try to implement pointer events in our three example scripts:

1. Drag and drop is a perfect match. Whether users employ a mouse or touch, or even a pen, they'll pick up an element, drag it along, and drop it. Pointer events would certainly increase the future-friendliness of a drag and drop script, since it's likely we won't have to add any code for the Kinect, TV remotes, and other pointer-like interactions.

2. The scrolling layer could work with pointer events. In touch interaction mode it will function perfectly. When using a pen, the user will press the pen on the layer, scroll it, and release it. That makes sense as well. The mouse interaction is similar to the pen, but picking up a layer by depressing the mouse button feels weird, and that's why I feel the effect is hard to port to the mouse. Still, the problem persists whether we use separate mouse and touch events or integrated pointer events, so pointer events don't do any harm.

3. Drop-down menus are the most complicated example. `pointerover` and `pointerout` seem to be made for this use case, but it turns out they aren't (see below). Drop-downs just don't work very well with touch interaction, and switching from touch to pointer events doesn't change that. The best way of handling drop-downs for touch screens is using the `click` event.

The examples show that pointer events work best if an interaction is not specifically tailored to one interaction mode. Still, the fact that the `pointerType` property, which we'll discuss later, tells you what kind of pointer you're dealing with, allows you to deal with mouse and touch separately, if you wish. (The property goes against the philosophical grain of the pointer events, but it's a practical necessity.)

The mouseover/pointerover Problem

`pointerover` captures old-fashioned `mouseover` events and what we could call "`touchover`" events: the user's touch, which is already on the screen, enters a certain element. `pointerout` does the same, but for when the user's finger or the cursor leaves an element. This is the `:hover` problem we discussed in the CSS chapter but in a JavaScript context.

Again we encounter the fundamental difference between continuous and discontinuous events. When moving from A to B with a mouse, the user has no choice but to enter and leave all intervening elements. When using touch, however, the user could touch first A and then B, and the only reason they wouldn't do that is because they're already dragging something. `pointerover` could be useful in such situations: is the element the user just entered a valid drop target?

Despite this scenario, `pointerover` remains fundamentally different from `mouseover`. Subtle `mouseover` effects, especially showing extra information, won't work on touchscreens because in such situations the user's touch is unlikely to enter the element from somewhere else. Instead, the user will likely touch the element without a preceding `pointerover`, so the extra information will not be shown.

The solution here is to stick with `mouseover` and `mouseout`, since they will fire during the touch interaction, just not when the user's touch enters or leaves an element. We'll get back to the details later. Still, this solution is not perfect since hovering remains an alien concept in a touch environment.

Progressive Input Enhancement

Just as responsive design taught us to design for many screen sizes, we have to find ways to design for many input modes. Let's call it *progressive input enhancement* for now. Unfortunately, progressive input enhancement is a lot less clear-cut than responsive design.

Many thanks to Jason Grigsby for clarifying these concepts in my mind. I borrowed the term progressive input enhancement from him, as well as several key thoughts in this section.

While responsive design is based on the idea that one design can adapt itself to all screen sizes, in many cases progressive input enhancement requires us to write separate scripts for the input modes — see, for instance, the scrolling layer example, which essentially needs three separate scripts for mouse, keyboard, and touch.

Besides, while screen size usually doesn't change during the user's interaction with a site, input mode could very well change. A Microsoft Surface user may start their interactions with the mouse, switch to touch, and switch back to the mouse again, all without leaving the page. Your scripts have to be ready for that. And yes, that's complicated.

Usually, this is important for tablets, but less so for smartphones, where the user has fewer choices of input modes. Still, assuming users will use one particular input mode during their entire interaction with a site is a consensual hallucination. It makes our jobs as web developers easier, but it has nothing to do with murky reality.

Responsive design can teach us a thing or two about progressive input enhancement. When creating a responsive design, it's a good idea to start with the most restrictive use case: the smallest screen. Progressive input enhancement will likely work the same. The most restrictive use case is likely to be the D-pad, which consists of four arrow keys and usually an OK or Enter button in the middle. The good news is that D-pads generally fire key events and use the same key codes as the arrow and Enter keys on a regular keyboard, which means distinguishing D-pads from regular keyboards is not necessary. The bad news is that they're still pretty restrictive.

This is the most restrictive input mode, and it would be a good idea to start designing your interactions here.

The worst news is that I don't have any particular guidance to share. Progressive input enhancement is so new that we haven't yet figured out strategies that will work most of the time, and even bright ideas that could help you further are scarce. You could see this as a problem, but also as a challenge. Who knows, maybe it's you who'll teach the world how to implement progressive input enhancement.

Finding the Current Interaction Mode

Possibly, progressive input enhancement will require you to detect the user's current interaction mode. Technically it's possible (though surprisingly hard) to do so, but the real question is what kind of useful information it would give.

Take, again, a Microsoft Surface user. You could detect that the user is currently using the mouse. But will that mean that they will continue to do so for the entire session? Not really. In fact, it seems quite likely that they'll use keyboard or touch at least occasionally. Or they may fold away the keyboard (with mouse trackpad) and go touch-only. If any of those things happens, what is the value of your interaction mode detection?

The only thing you can be reasonably certain of is that when users start a specific action in a specific mode, they won't switch mid-way. So if you detect a user moving a mouse for drag and drop, it's unlikely that they'll switch to touch mid-drag. But once the drag and drop is done, the user may elect to switch to touch for their next action — or stick with the mouse, or even go to the keyboard.

The most important thing you must do is make sure that all your interactions work for all interaction modes. Drag and drop should work for mouse, touch, and even keyboard. Once you've made sure it does, it doesn't matter any more which interaction mode the user is using.

But let's suppose you have good reasons for finding the current interaction mode. Maybe you want to gather statistics to see which mode the users are likely to employ. So let's go through a few cases:

- Pointer events are the easiest: they have a `pointerType` property whose values can be `mouse`, `touch`, or `pen`. Find the current value and you know what the user is doing.

- Another easy one: if any key event fires, the user is sure to use the keyboard. That doesn't say anything about future interactions — the user may switch to mouse or touch at any time. Still, it gives some useful information.

- Similarly, if a touch event fires you're certain that your user is currently using the touch interaction mode. Again, that doesn't say much about future interactions, but it's something.

- Watch out for mouse events: they also fire when the user touches the screen and are thus unsuited to detecting interaction modes. Detecting mouse use is a matter of ruling out all other interaction modes. If the user doesn't use touch or keyboard, it's likely a mouse is being used.

There are several ways of detecting the availability of a touch interaction mode. A method popularized by Modernizr, but unfortunately not quite reliable enough, is the following:

```
var hasTouch = !!('ontouchstart' in window)
```

If the `window` object has an `ontouchstart` property the browser supports the touch events and we can safely use them. At least, that's what you'd think. Although the first conclusion is true, a browser that supports the touch events doesn't necessarily run on a touchscreen device. The BlackBerry 6 browser, for instance, supports the touch events even if it runs on a non-touch device. Older Chrome versions had the same problem. And relying on `touchstart` leaves IE entirely out of the picture.

The only safe way of detecting touch is to see if an actual touch or pointer event fires. Only then you are sure that the browser supports touch and the user is currently using it.

```
var hasTouch = false;
document.ontouchstart = function () {
       hasTouch = true;
}
document.onpointerdown = function (e) {
       if (e.pointerType === 'touch') {
              hasTouch = true;
       }
}
```

It's best to go through the interaction modes one by one and see if you discover anything useful. Start with pointer events, since the `pointerType` property will give you usable information. Detect touch events next, since, as we'll see later on, a touch action also triggers mouse events, so mouse events should only count if touch is not detected. Keys come last — not for any fundamental reason but because they have to go somewhere. You could do something like this:

```
var interactionMode;
document.onpointerdown = function (e) {
       interactionMode = e.pointerType;
}
```

```
document.ontouchstart = function () {
	if (!interactionMode) {
		interactionMode = 'touch';
	}
}
document.onmousedown = function () {
	if (!interactionMode) {
		interactionMode = 'mouse';
	}
}
document.onkeydown = function () {
	if (!interactionMode) {
		interactionMode = 'keyboard';
	}
}
```

The ideal spot to run this check is a login screen or similar point where all users know they must interact with the site. While users log in, you use a script like the one above to detect which interaction mode they're using. Again, this doesn't tell you anything about a user's entire interaction with your site; only about their interaction with your login screen. But it's something, and it may have some predictive power.

Still, even this method cannot predict which interaction mode the user is going to use next. By far the best way of handling different interaction modes remains to code for all of them to make sure mouse, keyboard, and touch users can all use your interface.

The Touch Action Event Cascade

That was a lot of theory. Let's move on to practical stuff. The next few sections will investigate several aspects of touch events, and most of them also apply to pointer events. The chapter closes with a formal introduction to pointer events.

It's clear when `touchstart`, `touchmove`, `touchend`, `pointerdown`, `pointermove`, and `pointerup` fire. What's less clear is what to do with the mouse events. Despite them having no meaning on pure touchscreen devices, they're still vital to a lot of websites and even touchscreen browsers should fire them.

A few definitions so that we all know what we're talking about:

Action

An action the user takes; for instance, touching an element or swiping up.

Event

A specific JavaScript event that fires in response to the user action.

Event cascade

A series of JavaScript events fired in response to one user action.
The single-tap action, in particular, causes a long event cascade.

Event handler

A snippet of JavaScript that is executed when a specific event fires.

So a touch action leads to the firing of a cascade of events, and you can attach an event handler to one or more of them. (I advise you to usually restrict yourself to one event per cascade.)

That's why all browsers fire the mouse events just after the touch events. This leads to the *touch action event cascade*: the series of events that are fired when the user performs a touch action. Exactly which events fire depends on the action the user takes, as well as on the browser.

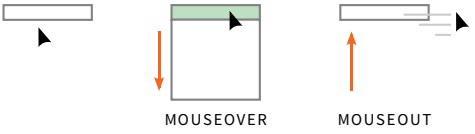
The Tap Action

When the user taps on an element, the following events fire:

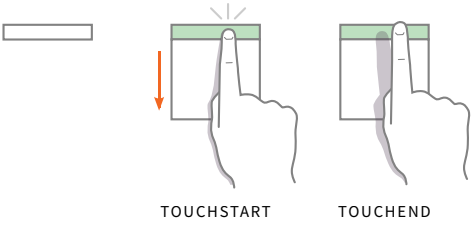
1. `touchstart/pointerdown`
2. `touchend/pointerup`
3. `mouseover`
4. one `mousemove`
5. `mousedown`
6. `mouseup`
7. click
8. Any `:hover` styles are applied to the element

The exact event order is not set in stone. Android WebKit 4, for instance, fires `mouseover` and `mousemove` before the `touchstart` event. Older BlackBerries, as well as Nokia's Symbian WebKit, fire `touchend` around the end of the cascade instead of at the beginning. All these differences do not really matter. In general, you create an event handler for just one of these events, and the handler will be executed when the user touches the screen, whatever the exact event you use. It's only when you use several mouse events that you can expect problems. The solution is to use at most one mouse event.

When the user subsequently taps on another element, the `mouseout` event fires on the original element and any `:hover` styles are removed. When the user subsequently taps on the same element, the entire cascade fires again, except for the `mouseover`. Apparently, the browsers assume the mouse (as it were) is already over the element. That doesn't entirely make sense, but it's the least bad way of implementing the mouseover concept on a touchscreen.



The traditional dropdown menu opens `onmouseover` and closes `onmouseout`.



When the user touches the drop-down menu, the `mouseover` event fires in the cascade and the menu opens. When the user removes her touch, though, the `mouseout` event does not fire and the menu stays open. This is deliberate: the user can now touch one of the links in the menu. Touching an element somewhere else on the page fires the `mouseout` event on the dropdown menu, and it closes.

Take the drop-down menu: the user touches a menu item; the event cascade fires, including `mouseover`; the script reacts by folding out the submenu. Now the user touches something else on the page: The `mouseout` event fires, and the submenu folds in. This is not quite the same as what happens with mouse interaction, but it's the best touchscreen browsers can do.

Other Actions

When the user does something other than tap, the event cascade is quite different. The `touchmove` and `pointermove` events now come into play, as do interaction-specific events such as `scroll` and `resize`. The mouse events are generally suppressed. In theory, if the user swipes over

a drop-down menu item, it will not open since `mouseover` is not fired. Generally speaking, these are the event cascades:

1. Swipe action: `touchstart`, `touchmove`, `touchend`, `scroll`
2. Pinch-zoom action: `touchstart`, `touchmove`, `touchend`, `scroll`, possibly `resize`
3. Double-tap action: `touchstart` and `touchend` twice, then `scroll` and possibly `resize`
4. `Touchhold` action: `touchstart` and `touchend`; in a few browsers `contextmenu`.

Only Safari, Blink, BlackBerry 10, the Nokia browsers, and Firefox follow these event cascades most of the time. The other browsers diverge from them, but, again, that matters little as long as you use only one non-touch event. IE fires the appropriate pointer events instead of the touch events.

Some browsers fire more events. In particular, Android WebKit 4 always fires `mouseover` and `mousemove`, so drop-down menus will fold out when the user swipes or pinch-zooms. Although this may be surprising, it will hardly break your site. In most practical situations these browser differences do not matter a lot.

The `contextmenu` event fires when the user right-clicks in a desktop browser. The mobile equivalent is `touchhold`; that is, keeping your finger in place for about a second. Unfortunately, few mobile browsers support the `contextmenu` event, so make sure your interactions don't depend on it.

The Viewports chapter has more details on the `resize` event.

Safari: Canceling the Cascade

Now we come to two Safari oddities. Safari has its own take on the event cascade: if a `mouseover` or `mousemove` event causes a content change, Safari cancels the rest of the cascade and does not fire the `mousedown`, `mouseup`, and click events. There are two questions here: why `mouseover` and `mousemove`, and what does a content change mean?

`mouseover` is simple: it's because of drop-down menus. This is the solution Safari devised for the problem of a menu item being a link. `mousemove` is murkier — it seems that some sites use it in comparable situations, but I'm not sure of the details and suspect that these sites are doing it wrong. (That doesn't matter, though; a mobile browser must accommodate even wrong sites.)

It turns out that a content change means a DOM change, but only if actual DOM methods such as `appendChild()` and `insertBefore()` are used. An `innerHTML` change does not count. Most style changes, notably a change in `display`, count as DOM changes — again specifically because of drop-down menus.

Safari: Mouse Event Bubbling

The other Safari oddity is that mouse and click events only bubble up to the document in specific conditions. I'm pretty sure this is deliberate behavior even though I don't understand why it was introduced.

All events in the cascade should bubble up, and do so in all browsers but Safari. In Safari, the mouse and click events only bubble up when one of the following conditions applies:

1. The target element of the event is a link or a form field.
2. The target element, or any of its ancestors up to *but not including* the `<body>`, has an explicit event handler set for any of the mouse events. This event handler may be an empty function.
3. The target element, or any of its ancestors *up to and including* the document has a `cursor: pointer` CSS declaration.

I used to think that this was a performance trade-off — that something in the bubbling of mouse events (which takes place every time the user touches the screen) would slow down the system or eat too much battery power. But if that's true, why don't any of the other browsers implement it? Besides, just moving up the DOM tree and checking if elements have event handlers attached to them doesn't take that much time and energy, does it?

> Event bubbling is the process by which an event moves up the DOM tree from its target until it reaches the document, and executes any event handler it finds on the way. You can set one general `mouseover`, `touchstart`, or `click` event handler on the document and all these events on the entire page end up with that handler. Most events bubble, and all browsers support it.

Still, this is how mouse event bubbling works at the time of writing. So if you run into bubbling trouble in Safari, if you expect a mouse or click event to end up at the document or the body level but it mysteriously doesn't, add an empty event handler to the event's target element. This is enough:

```
targetElement.onclick = function () {}
```

Bubbling now works for all events. Weird but true.

Anatomy Of A Click

The most important event in the touch cascade (and arguably in JavaScript as a whole) is the `click` event. It always fires when you activate an element; that is, when you use an element such as a link or a button for its intended purpose: following a link, submitting a form, or whatever. It doesn't matter how you activate the element — the `click` event fires when you click a mouse button, tap the screen, or use your keyboard.

`click` is really a misnomer: the event should have been called `activate`. However, it got its current name because it was invented with the mouse events, and by now it's far too late to change that.

The best news is that not only all current browsers support the `click` event, but all future browsers will as well. Activating links and buttons will always be possible, and too many websites depend on the `click` event for any browser ever to consider dropping it. My golden rule is: **stick with click**. If your website uses only the `click` event you do not need to change anything, no matter what kind of devices hit the market.

Still, there are two problems you will encounter on touchscreen devices: the infamous lag and, very rarely, severe difficulties in getting a `click` event to occur.

An useful overview of the problem and several possible solutions can be found at http://smashed.by/mwhb12

300 Milliseconds

There is a 300 millisecond lag between touching an element and that element doing whatever it's supposed to do. Fundamentally, this problem is unsolvable because the act of touching the screen is overloaded. At the moment a finger touches the screen the browser cannot know if the user

wants to tap, double-tap, swipe, hold, or do something else. The only safe course of action is waiting a little while to see what happens next.

It's especially the double-tap that is the problem. Even if the browser detects your finger leaving the touchscreen it still doesn't know what to do. Will your finger return for another tap, or was this the entire action and should it fire the single-tap cascade? In order to be sure it must wait a little while, and browser engineers found out that the best value for that little while is 300 milliseconds.

Still, both web developers and users find the delay annoying. Therefore browser vendors are searching for circumstances where it's safe to assume a double-tap will not take place — in other words, when users won't have to zoom. If that's the case, they don't have to implement the delay. Many browsers remove the delay when a page is made unzoomable, but disabling zooming is evil and I actively discourage it.

Chrome takes a more interesting approach. If the author sets a meta viewport with `width=device-width`, Chrome is willing to assume that a double-tap action will never take place since zooming is unlikely to be necessary. In these conditions it does not wait for a potential second tap but fires the tap event cascade straight away.

> We've discussed `width=device-width` at length in the Viewports chapter.

It is important to realize that the Chrome team is making a guess here. Sure, the theory sounds fine, but it may be wrong regardless, and the solution may turn out not to work. Nonetheless, it's the best solution I've heard so far, and I'm glad Chrome is conducting this experiment. If it works we'll see other browser vendors copying it.

IE allows you to suppress the delay by using `touch-action: manipulation`. Now double-tap zooming is not possible, and the browser can remove the delay. We'll get back to this bit of CSS later.

One browser that will never remove the delay is Safari, because on iOS a double-tap is also a scrolling command. Since Apple never wants to remove scrolling ability, it won't be able to create circumstances in which it's safe to remove the delay.

Don't try to solve this problem on your own. Technically, it's trivial to execute your event handlers `ontouchend` instead of `onclick`, but that would mean you would get a weird situation when the user double-taps in order to zoom. You can work around that, too, if you wish, but after spending several sleepless weeks on the script you'll find that the best solution is to include a slight delay after the first `touchend` event. Back to square one; you've wasted your time. The delay is here for a reason.

The Same Pixel

Once upon a time there was a phone built by an important device vendor that shall remain nameless, for an important operator that shall remain nameless. My job was to test a browser created by the vendor, and so I installed a bunch of test widgets I'd created (widgets are a now-obsolete type of web app). Many of those widgets required me to click a button, after which the test would take place.

I couldn't click the buttons. That is, I could touch the screen as often as I wanted, but that rarely translated to an actual `click` event or activation of the button. I spent minutes and minutes rapidly tapping the screen in the hope that the browser would allow at least one click.

Sometimes that worked, but by that time I had usually forgotten what I wanted to test and stared blankly at a bunch of random numbers.

It took me a while to figure out what was going on. On desktop, a `click` event fires when a `mousedown` and a `mouseup` event take place on the same pixel — when the mouse cursor does not move between the depressing and releasing of the button. This works fine: a mouse cursor always points to one single specific pixel, and in order to point to a different pixel you have to consciously move the mouse.

That's not how it works on touchcreens. Your fat finger touches the screen, inadvertently moves a few pixels, and releases the screen. As a result the `touchstart` and `touchend` events do not take place on the same pixel, and this particular browser decided a `click` event was not called for.

Good mobile browser vendors give users some leeway in this situation. They allow a `click` event to take place even when the `touchstart` and `touchend` didn't take place on exactly the same pixel. My research shows that they allow a finger to move between 4 and 20 pixels, depending on the browser, before deciding that a `touchmove` is taking place and `click` should not be fired.

You'll occasionally encounter this problem in beta browsers — for example, I caught traces of it in early builds of Firefox OS. You now know what the problem is, and although you can't solve it (there is no solution), at least you can impress your peers by your detailed knowledge of mobile click events.

Taking Apart The Touch Events

If you listen to touch or pointer events, you want to know more about them. Specifically, you want to know the target or the coordinates of the touch event. That means digging into the event object, just like with any other JavaScript event.

```
var el = [an element];
el.addEventListener('touchstart',handleTouch,false);

function handleTouch(e) {
	// e refers to the event object
	// e.type gives the event type
	// e.target gives the event target
	// return false or e.preventDefault() prevents
	// the default action
}
```

`Type`, `target`, and preventing the default action work like with any other event. There is more to say about preventing the default action, and we'll get back to it later, but you need a standard `return false` or `e.preventDefault()` to begin with.

touchLists

There's one special feature that touch events have that is lacking in pointer events: touch event objects contain references to `touchList` arrays that hold objects for every individual touch. If the user is currently using four fingers you will find four entries in a `touchList`. There are three `touchLists`.

1. `touches`: a list of currently active touches on the entire screen

2. `changedTouches`: a list of touches whose change led to the firing of the touch event.

3. `targetTouches`: a list of touches that started on the event's target.

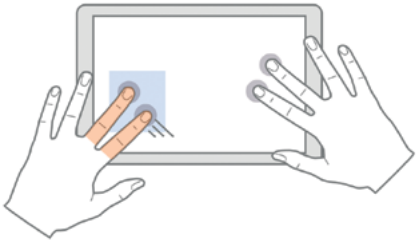
The trick is to use `changedTouches`, and the reason for that is the `touchend` event. If the user lifts their last finger from the screen and causes a `touchend` event, the touch doesn't exist anymore and doesn't appear in the `touches` or `targetTouches` lists. But since it was the removal of this touch that caused the `touchend` event to fire, `changedTouches` still contains information about the removed touch. That's vital if you want to know where a drag-and-drop action ended.

These `touchLists` work as arrays in that they have indexed entries, and every entry is an object that contains interesting information about one touch, notably the coordinates.

A user has four fingers on the screen, of which two are on the same element. She moves one finger, and a touchmove event fires. Which touches are included in the three touch lists?

On older screens, and with zoom 100%, one CSS pixel equals exactly one device pixel.

The `changedTouches` list contains only the touch that caused the event: the moving finger.



The `targetTouches` list contains only the two touches that are on the element.

Finding the Event Coordinates

This is how you find a touch event's coordinates:

```
function handleTouch(e) {
        // use pageX/Y instead of clientX/Y if necessary
        var touch = e.changedTouches[0];
        var coorX = touch.clientX;
        var coorY = touch.clientY;
}
```

The first object in the `changedTouches` list is the touch that caused the event that just fired. (Usually, the `touchList` doesn't contain any other objects.) This object contains coordinate information stored in `clientX/Y` and `pageX/Y`. Other coordinate properties, notably `screenX/Y` and `x/y`, have serious browser issues and should not be used.

However, we don't have a true cross-browser script yet. First, IE doesn't support `touchLists` and gives coordinates the old-fashioned way on the event object itself. Second, in cases such as drag-and-drop scripts it would be useful if we had one function that finds coordinates for both mouse and touch. So the 100% safe cross-browser compatible function for finding event coordinates is:

> The difference between `clientX/Y` and `pageX/Y` is that the first pair is calculated relative to the top-left corner of the visual viewport, but the second relative to the top-left corner of the layout viewport, which may have scrolled out of view. Which one you should use depends on which coordinate set makes most sense for your script.

```
function findCoordinates(e) {
        // use pageX/Y instead of clientX/Y if necessary
        var x,y;
        if (e.changedTouches) {    // touch events
                x = e.changedTouches[0].clientX;
                y = e.changedTouches[0].clientY;
        } else {            // pointer or mouse events
                x = e.clientX;
                y = e.clientY;
        }
        return [x,y];
}
```

Leaving the Element

Say you bind `touchstart`, `touchmove`, and `touchend` events to a certain element. If your finger leaves that element, `touchmove`, and in some browsers `touchend`, will continue to fire. That is, as long as the touch starts on the element, `touchmove` continues to fire as long as the touch remains on the screen, even if it leaves the element the event handler is defined on. The opposite is not true: if you start your touch elsewhere and then move your finger to the element, `touchmove` does not fire.

We now see why `touchenter` and `touchleave` events would be useful. The current touch events don't give any hint of the touch leaving the element they were defined on, and occasionally that information is necessary.

This behavior does not occur with the pointer events. When your finger leaves the element, `pointermove` stops firing.

Preventing the Default

Any event handler allows you to cancel the default action of the event by returning `false` or calling `event.preventDefault()`. Doing this `onclick` makes sure the link is not followed, `onsubmit` it cancels the sending of the form, and so on. This is ancient and reliable functionality. It also works for the touch events, but not for the pointer events.

First, a note: some devices do not allow the default of some gestures to be canceled. For instance, the iPad will never cancel the default of a four-finger swipe, since it switches from one app to another and is considered a fundamental OS-level interaction. Scripts are not allowed to touch it.

If you prevent the default `ontouchstart`, the browser concludes that you do not want the default action associated with your finger movement to take place. Thus, a link is not followed, a swipe does not result in a scroll, and a double-tap does not zoom in or out. Also, preventing the default of a touch event prevents the cascade of mouse and click events.

The only slightly tricky bit is distinguishing between `touchstart` and `touchmove`. It turns out the browsers handle this quite sensibly: if the default action requires both `touchstart` and `touchmove`, such as scroll, a `return false` on either will cancel the scroll. If the default action only requires a `touchstart`, such as a tap or double-tap, you must `return false ontouchstart`. `Touchmove` will not help you here.

1. `touchstart` only: tap (click), double-tap, touchhold, and the event cascade.

2. `touchstart` or `touchmove`: scroll or pinch-zoom (though Android WebKit 2 doesn't allow you to cancel a pinch-zoom at all).

The pointer events, and therefore IE, don't support `preventDefault()` at all. It is not possible to prevent the default action in JavaScript — instead you have to use the CSS `touch-action` declaration that we'll examine later. This is actually a feature, and not a bug. The IE team wanted to make sure that any direct manipulation of an element (that is, performing any touch action on it) would have immediate, visible results. If you allow `preventDefault()`, it might take too long for the instruction to reach the browser due to the script being slow. That might cause a visible stutter: initially the element reacts to the touch action, but after half a second or so it doesn't anymore because the default is canceled. This is the main reason pointer events moved default canceling from JavaScript to CSS.

Example: Horizontal and Vertical Scrolling

The scroll script contains an example of default canceling. I decided that a horizontal scroll swipe should cause the element to scroll, while a vertical one should cause a regular page scroll. How do I make sure that happens? By canceling the default for horizontal scrolling, but not for vertical scrolling.

This is what I do. If the movement is mostly vertical (*y* is larger than *x*), I want the page to scroll normally, so I must `return true` in order to hand the event over to the OS. If the `touchmove` is mostly horizontal (*x* is larger than *y*), the script should move the scrolling element and the OS should do nothing; I `return false` to cancel the default scroll action.

```
document.ontouchmove = function (e) {
        // origin[] has the coordinates of the touchstart
        var currentPos = findCoordinates(e);
        var newPosX = (currentPos[0] - origin[0]);
        var newPosY = (currentPos[1] - origin[1]);
        if (Math.abs(newPosY) > Math.abs(newPosX)) {
                returnValue = true;
        } else {
                returnValue = false;
                // pos is the element's X-coor when the
                // scrolling started
                newPosX += pos;
                // min and max are the pre-calculated
                // minimum and maximum scroll
                if (newPosX <= max && newPosX >= min) {
                        layer.style.left = newPosX + 'px';
                }
        }
        return returnValue;
}
```

Pointer Events

It's time to focus on pointer events. They don't differ that much from touch events, but there are several slight incompatibilities that you need to know about. Also, you must know about the `touch-action` CSS declaration.

IE11 supports the pointer events detailed in the table below. The event names deliberately resemble the mouse event names: the pointer events are exactly the same, except that they work with any kind of pointer, not just a mouse.

Touch and pointer events differ in the following ways:

1. At the time of writing, pointer events are only supported by IE. Although Chrome and Firefox are considering implementation, they don't ship pointer events yet.

2. Pointer events need the MS vendor prefix and sometimes camelCase event names in IE10. See below.

3. Pointer events don't have `touchLists`; so event coordinates are found in the event object itself, just as with mouse events. We already discussed this problem and its solution.

4. It's not possible to cancel the default action of pointer events in JavaScript for reasons we've already discussed.

Event	Meaning	Bubbles	Support
`pointerdown`	The mouse button is depressed or the user's finger or pen touches the screen.	Yes	IE10
`pointermove`	The pointer moves on the element the event is defined on. If the pointer leaves that element the event stops firing.	Yes	IE10
`pointerup`	The mouse button is released, or the user's finger or pen leaves the screen.	Yes	IE10
`pointerover`	The pointer enters the element the event is defined on or one of its children.	Yes	IE10
`pointerout`	The pointer leaves the element the event is defined on or one of its children.	Yes	IE10
`pointerenter`	The pointer enters the element the event is defined on.	No	IE11
`pointerleave`	The pointer leaves the element the event is defined on.	No	IE11

Pointer Events

The difference between `pointerenter/leave` and `pointerover/out` is the same as between `mouseenter/leave` and `mouseover/out`. The `over/out` events fire when the pointer goes into or out of the element the event is defined on, or any of its children. The `enter/leave` events only pay attention to the element itself, and not to its children. Generally speaking, `enter/leave` are the events you want.

Names and Prefixes

In IE10 the pointer events are prefixed and sometimes camelCased. IE11 supports this as well, but also unprefixed lowercase event names. Microsoft warns that eventually the IE10-compatible names will be dropped. Thus, a future-friendly script that must also support IE10 requires you to use both systems. The following event names are the only ones that work:

```
el.onmspointermove = doSomething;
       // lowercase; IE10 and IE11
el.onpointermove = doSomething;
       // lowercase; IE11 only
el.addEventListener('MSPointerMove',doSomething,false);
       // camelCase; IE10 and IE11
el.addEventListener('pointermove',doSomething,false);
       // lowercase; IE11 only
```

If you decide to drop IE10 you can use the IE11-specific ones only: they'll continue to work forever. If you must support IE10 you have to combine the event names somehow.

Also, the event type changed from IE10 to IE11. Where in IE10 it was `MSPointermove`, in IE11 it has become `pointermove`. This is sometimes annoying, since if you want to do something when a certain type of event is received, you should write:

```
function doSomething(e) {
        if (e.type === 'MSPointerMove'
                || e.type === 'pointermove') {
                // this is a pointermove event
        }
}
```

Again, if you drop IE10 you can just check for `pointermove` and be done.

Event Properties

Pointer events have the usual event properties such as `target` and

Property	Value	Meaning
`pointerType`	mouse, touch, pen	What type of pointer is being used
`isPrimary`	true or false	Whether the current pointer is the primary pointer. In practice, the mouse pointer and the first touch pointer to touch the screen are primary pointers; the rest aren't.
`pressure`	0.5	How much pressure is being exerted, on a scale of 0–1. Since the current crop of devices can't read pressure it's always 0.5. That may change in the future, though.
`pointerId`	integer	The unique ID for each pointer. The mouse pointer is always 1, but each touch pointer gets its own ID.
`width and height`	1	The size of the pointer, in (probably) CSS pixels. Right now, devices can't yet read that out, so both are 1, but future devices may give the actual width and height of the fat finger you apply to the screen.

Event Properties

`clientX/Y` — remember that coordinate properties are defined directly on the event object, and not in a `touchList`. We've already covered the cross-browser function for finding event coordinates.

Pointer events have a few interesting extra properties, though. `pointerType` is one you may want to use right away, while the others are more like forward-looking properties that don't make much sense right now but may become important one day.

touch-action

Finally, there is the `touch-action` CSS declaration. It tells the browser which kind of touch actions should be handled by the OS. It is supported by IE, and Chrome support is expected to land in summer or autumn 2014.

`touch-action` allows you to make a distinction between actions meant for the OS and actions meant for your script. For instance, the scrolling layer script listens only to horizontal panning, and leaves vertical panning to the OS. We already saw how to handle that in JavaScript, but since pointer events don't support `preventDefault()` we must use `touch-action: pan-y` instead. This tells the browser to handle vertical pans normally but suppress the defaults of all other touch actions.

Like with event names, IE10 uses a vendor prefix, while IE11 allows both a prefixed and an unprefixed version. If you want to support IE10 you must use `-ms-touch-action`, while the future-friendly version is `touch-action`. These properties take the following values, and note that every value tells the browser which actions are handled by the OS:

touch-action:	Meaning
none	OS does not react to any touch action
auto	OS reacts to all touch actions
pan-x	OS reacts to horizontal panning
pan-y	OS reacts to vertical panning
pinch-zoom	OS reacts to pinch-zooming
double-tap-zoom	OS reacts to double-tap zooming
manipulation	OS reacts to everything except for double-tap

Touch actions

You can combine values, so that `touch-action: pinch-zoom pan-y double-tap-zoom` makes the OS handle double taps, pinch-zooms, and vertical swipes, but suppress `pan-x`. This is in fact the value we need for the scrolling layer script.

It's probably not a good idea to use `none`. Since it suppresses all OS handling of touch actions, you will be responsible for creating custom alternatives for scrolling and zooming. You have better things to do. Only use `none` if you're absolutely certain you want to take over the entire handling of touch actions. In other cases, combine values to describe exactly what you want the OS to handle.

The `manipulation` value is specifically for suppressing the 300 millisecond delay between the touch action and the `click` event, so it might be useful to sprinkle that through your code.

***

We have now explored the CSS and JavaScript features that are unique to mobile. What remains is some general advice on how to become a mobile web developer. And that's exactly what the next chapter of this book will consider.

Chapter 7

Becoming A Mobile Web Developer

Chapter 7

Becoming A Mobile Web Developer

Whew. We've learned a lot about the mobile web throughout this book. I assume you've started to get a feel for viewports and touch events, `:hover` and browsers. It's time to wrap things up with a quick overview of what it takes to become a mobile web developer.

What exactly is a mobile web developer? By now most web developers will have looked at their sites on their own mobile phone, and even solved an iOS or Android bug or two, but that's not enough. To me, a mobile web developer is someone who spends a lot of time on mobile browsers, and for whom Android WebKit compatibility is as important as IE compatibility.

The most important advice I can give you is to start doing mobile testing right now. You probably have your own iPhone or Android device: use it to look at your current project. Now. Also, download Opera Mini and test in that browser, too. If you have an Android device, download a few more browsers and test in them, too. I advise Chrome, Firefox,

and UC. All of them are available in Google Play — though, as we saw in the Browsers chapter, they don't exist for any of the other platforms. (Remember: Chrome on iOS actually uses the Apple WebView, and not Blink.) True, your single device is not representative of the market as a whole, but any mobile test is better than no mobile test. Besides, even one single browser on one device allows you to get acquainted with the small screen and responsive web design.

The Ideal Device Lab

Your first job will be to create a device test lab. Here's the ideal lab as of summer 2014:

1. At least half your device lab will be Androids. We'll get back to them below.

2. iOS: at least one iPhone and one iPad; possibly also an older iPhone (with less capable hardware) or an iPad Mini (with a smaller screen). It's useful to have one device on an older version of iOS; a few users won't upgrade, either because they don't want to or because their device is too old. In fact, it's useful to have an old device yourself, so that you're sure your site will also run in adverse conditions. Also, make sure you have one Retina device for those resolution and responsive images tests.

3. BlackBerry: especially in the UK, where BlackBerry still has a browser market share between 5 and 10%, it's important to have one or more of them for testing purposes. I'd say one BB10 device (Z10 or Q10 or even newer), and an older one with OS6 or 7. If you can get your hands on a non-touchscreen one that would also be useful.

4. Nokia is complicated because it currently supports three different platforms with varying levels of global market share. What you will need in any case is a Windows Phone, preferably a new one. Windows Phone has a fairly low market share but it's slowly rising and you do want to test in IE Mobile. If you have enough money, buy a Windows Phone 8 (with IE10 Mobile) and a Windows Phone 8.1 (with IE11 Mobile); increase both by one version number if IE12 Mobile is available by the time you read this.

5. As for the other Nokia platforms, that depends on whether you will do a lot of work for Africa, Asia, or Latin America. If you do, it's good to have an Asha (S40) with Nokia Xpress handy, since they are still being used a lot. Symbian is a dying platform. If you happen to have a Symbian phone or can get one cheaply, add it to your line-up. If you don't, don't bother. (By the way, even at its zenith Symbian was not important in the US.)

6. Windows 8: at least one Windows 8 tablet: either a Microsoft Surface or one from the other vendors. Windows 8 tablets are different from phones, both in support (no meta viewport, for instance), and in the fact that they support touch, mouse, and keyboard and you can use the three interaction modes interchangeably.

7. Then, the minor OSs: Firefox OS, Tizen, Amazon Kindle, and Sailfish by Jolla. They don't have market share to speak of at the time of writing, but that could change. Keep an eye on them and add them to your lab when necessary — and if you can get one on the cheap, do it. (At the time of writing Firefox OS phones are especially inexpensive.)

Updates

Be careful with browser or OS updates, since they can disturb your test setup. I never update anything while going through a test, but since I run compatibility tests and you will likely run website tests, your mileage may vary. Set a rule at the start: either update everything straight away, or postpone all updates until your current test run is finished.

As we saw in the Android chapter, Android has a complicated update timetable. The other OSs are usually less hassle. iOS users, in particular, tend to update their devices quickly, although a small minority will not be able to update due to their device being too old.

I alternate updates between my iPhone and iPad. At first my iPhone ran iOS5 and my iPad iOS6, and when iOS7 was released I updated my iPad, but not my iPhone. When iOS8 is released (a few months after writing this) I'll update my iPhone, but not my iPad. Thus I always have the latest two versions available. You should establish a similar rule for your iOS devices.

As for downloadable browsers, they will update far more frequently than device OSs. Install every update, just as you would on desktop.

Android

As we saw in the Android chapter, the main characteristic of Android is its differentiation. Whenever you buy an Android device, make sure it comes from a different vendor and has a different screen size and Android version than the devices you already own. You could make an exception for the Samsung Galaxy range, or any other model that has a very high market share at the time you read this.

Once you've bought the device, test the default browser carefully to establish its identity and make a note of it. Remember: any browser that has `Chrome` in its user-agent string (`navigator.userAgent`) is Chrome, though not necessarily Google's Chrome, while any default browser that hasn't is Android WebKit.

I advise you to read through the long browser list in the Android chapter again, since one of your purposes is to have at least one of each default browser available, and two or three Android WebKit 4s.

So a good Android lab consists of the latest high-end Samsung Galaxy, maybe an earlier high-end Galaxy, an HTC, a Sony, and an LG. Grab a Chinese Android (preferably a Lenovo, Huawei, or Xiaomi) if they are available in your market, and add Motorola to the list if you're in the US. Make sure at least one of these devices is a mid-range one (say €100–150), and at least one runs Android 2.

Owning a Google device is not a top priority. Despite their popularity among web developers, normal consumers don't buy all that many of them, and the Google Chrome that runs on these devices is not representative of Android default browsers of other vendors. I advise you to postpone buying a Nexus until you have 3 or 4 non-Google devices.

Once you have these devices, install the other browsers on them: Chrome, Firefox, Opera Mobile, Opera Mini, and UC are the important ones, but as long as you're at it you should add UC Mini (proxy browser; very popular in China), QQ (also called One; Chinese), Puffin (Korea), and anything else you can lay your hands on. The purpose is not so much making sure your site runs perfectly (though that is a definite

bonus), but getting acquainted with the odd things mobile browsers may do to your site. And all these browsers are free.

Spread out these browsers across your Android devices — don't install them all on one device. Once you get to the actual testing you want to be able to use several devices side by side.

No Experimental Versions

Even though you are a power user, your job is to create websites that work for the average consumer. Be careful to test only on OS versions and in browsers that consumers actually use. It's nice to know that feature X is supported in version Y, which is in beta, but consumers don't use betas so you can't yet use feature X, despite it working on your device.

I advise you to only test in the latest and next-to-latest consumer versions of the browsers or OSs, and not in beta ones. This will serve your clients better and also keeps your tests simpler.

In particular, do not install custom versions of Android on your phones (called flashing the ROM). Consumers don't do that, so your device will be useless for testing. Keep the OS and browser exactly as they were when you bought the devices, and only install official updates. These are test devices, not personal ones.

Acquiring Devices

Knowing what an ideal device lab should contain is good, but how do you go about actually acquiring the devices? Although larger companies will be able to allocate a few thousand euros or dollars per year for devices, freelancers are usually not in that position.

The solution is simple: acquire them *slowly*. Save at least €100 per month to buy devices. This will allow you to buy two high-end or five mid-range devices per year. That's not really a lot, but it's better than nothing. The more you save per month, the better, obviously. Take a look on eBay and similar sites — sometimes you'll find nice second-hand deals there. A few scratches won't impede your testing.

You already have an iPhone or Android for your personal use. The first test device you acquire should be an Android if you have an iPhone, or an iPhone if you have an Android. The second should be another Android — buy one from a different vendor with a different screen size, a different default browser, and a different Android version. After that should probably come one or two more Androids, an iPad, and a device that's neither iPhone nor Android.

Occasionally, freelancers feel that acquiring devices will cost them a lot of money. Although that is correct in a literal sense, serious mobile testing requires a serious test lab. Just as you buy a few computers and a few software packages if you need them, you should buy devices as well. They represent a business expense that you hope to earn back (with interest) from your clients, since you can do a better job with them than without them.

Sharing Devices

Still, if you're a freelancer who can afford two devices per year, you'll initially have a lab of only three or four devices: two professional ones, your personal one and maybe your personal tablet. That's enough for a few simple design tests, but not enough if you want to use complex JavaScript and have to make sure the performance is acceptable everywhere.

Fortunately, you're very likely not the only one in your area with this problem. If you find other developers who are building a lab of their own, invite them to compare notes and swap devices. Not only will you be able to test your websites on more devices, you will also acquire useful contacts to discuss technical issues and the mobile market in general, and perhaps serve their surplus clients.

On http://opendevicelab.com/ you can find many open device labs around the world — maybe even one in your city.

All over the world, open device labs are becoming popular. Usually founded and supported by a small local company with a decent set of devices, these labs are open to anyone who wants to test mobile devices, provided they reserve a slot in advance. In return, you can leave your devices there if you don't need them for a while. Finding out whether your city has one is worth the trouble. If it doesn't, maybe you should set one up? If nothing else it could net you some valuable contacts.

What To Test

OK, so you have the perfect device lab. Now, what will you test? Obviously, you start with basic website behavior, just as on desktop. Do the CSS and JavaScript work? If not, how do you solve the problem? Do your responsive design breakpoints need adjustment? Maybe the two-column layout should kick in at a slightly larger viewport width? You can figure this out for yourself.

There are a few things that are different from desktop, though, so a really thorough mobile testing procedure includes the following:

1. In addition to Wi-Fi, test your site over a data network — ideally 3G or better, but also on 2G if many people in your target region can't get anything else. If you're really thorough, you test on several networks. A real network connection can lead to unexpected situations such as a blazing fast connection directly followed by no connection at all. (If you're really thorough, test in a moving train where circumstances change all the time.) What happens when the connection suddenly fails?

2. Test in both portrait and landscape orientation. As we saw in the Viewports chapter, the ideal viewport will change with the orientation in most browsers. That's not all, though. How do fixed layers behave? Maybe there are problems with repositioning or recalculating a specific element, especially modal windows and complex items such as image carousels. Open a modal window or use a carousel, switch orientation, and see how it behaves.

3. Test your interactions, especially when they involve custom gestures. Do all gestures work properly on all devices?

4. Form elements merit special attention, since usually they're linked to critical transactions such as paying for something. They have to work flawlessly in all browsers, both orientations, and when the user zooms in. Make sure to actually fill out the form and try to guess what the user will do once the software keyboard appears. Pay special attention to heavily styled form elements or components such as calendar widgets. Do they work in all browsers and orientations?

How To Test

You'll quickly find that testing your sites on mobile devices is much more time-consuming than on desktop, and not only because there are more mobile browsers. Below is some advice for mobile browser testing based on my own experience. Please don't get too hung up on the details; it's perfectly fine if you structure your testing differently. Novice mobile testers will find some useful hints here, though.

Time

Testing something on mobile devices takes more time than you think, even if you start out by assuming it'll take more time than you think. There is no such thing as a quick mobile test. Allow them to go way over time if necessary. Do not start a quick test 15 minutes before you're supposed to go home. You won't go home on time.

Preparing the Devices

There are certain preparations you have to make before starting the actual tests. The most important one is some sort of syncing solution. You want to be able to click through your site on your desktop computer, with the phones following along. This greatly cuts down testing time, since you don't have to perform every click on every phone. At the time of writing there are two major tools for syncing:

> For an overview of mobile testing see http://smashed.by/mwhb15, where Addy Osmani talks about several tools.

- Ghostlab, which requires you to add a script to your page. See http://smashed.by/mwhb16 for more information. Don't forget to remove the script once your tests are complete.

- Adobe Edge Inspect syncs iOS and Android devices to your desktop Chrome. You can find more information at http://smashed.by/mwhb15

On every device, set the screen timeout to its maximum value. This timeout is for switching off the screen after a period of inactivity, and switching the phone on before reloading your page becomes annoying after a while. The timeout can generally be found in the display settings of the device.

Add an icon to the home screen for every browser. You'll usually do this, but forgetting it even once can lead to problems down the line. If you can change the icon text, note the browser version.

Make sure all devices are charged. This sounds like a no-brainer, but I found that the only way to actually make sure is to be pretty disciplined about it. Plug in all phones in the next batch before starting on your current batch (see below for batches): this ensures that the phones are ready when you are. Another no-brainer: make sure you have multiple power sockets available. You don't want to charge just one phone, but up to eight or so.

Nowadays, nearly every device has Wi-Fi capability. Nonetheless, a few devices (notably Windows Phones) can be fairly slow in setting up a connection even to a known access point. Make sure to switch these phones on a minute or so before the actual testing starts. In case you test on non-Wi-Fi phones, make sure to insert a SIM card and start them up a few minutes before you need them.

Testing in Batches

Once you get beyond five or six browsers to test in, it becomes useful to divide them up into batches and test one batch at a time. Count every instance of Android WebKit as a separate browser, as well as every version of any browser. The purpose of batch testing is not to be overwhelmed by bugs and oddities, but to solve them one or two at a time.

Make an ordered list of devices and browsers you want to test. I encourage you to write down this list, including device names and exact version numbers of OSs and browsers. This will become an invaluable reference in the later stages of testing, when your enthusiasm is flagging and your true interest lies in throwing mobile devices around the room. By then you won't be able to remember what you should test next, and looking at your list will save you a lot of headaches.

Once you have that list, divide it into batches. The first batch should be a mixed one, while the rest should have a common theme (Android, Opera, and so on). The purpose of the mixed batch is to get a quick overview of whether your code is going to work across different browsers or not. If you find a lot of problems in the first batch, you should probably choose another approach. If you don't find many problems you're on the right track and you can continue with detailed tests in the other batches.

Each batch should contain between three to eight browsers. Make sure that in every batch each browser runs on its own device. You do not want to switch to different browsers on the same device since that takes way too much time and will become confusing after a while.

When you initially create a page you should test constantly in the mixed batch of about four to five browsers. This batch should contain one Safari, one Android WebKit (not Chrome!), one Opera Mini, one browser that's neither iOS nor Android, and maybe one or two other browsers that are important for your client's target audience.

Once your page works in this first batch, the time comes to test it in all browsers on your list, batch by batch. The problem here is that if you notice a problem and change the page, you have to go back to the beginning and test the new version in all previous batches. Therefore, it's best to start with the most problematic browsers that will likely need many adjustments: Android WebKit, IE, and the proxy browsers.

So a possible batch list could look like this; adjust to taste and device lab:

1. The mixed batch of one Safari, one Samsung Android WebKit 4 (not Chrome!), one Windows Phone, one Opera Mini, and one Chrome.

2. A batch of Android WebKits: all Android devices you have available. If you have more than one Android 2 device, do Android 2 first, then Android 4. Make sure these are all Android WebKits, and not Chromes.

3. A batch of IEs and proxy browsers: say IE10, IE11, Opera Mini on two or three devices, Nokia Xpress, UC Mini. By the time you finish this batch you'll have found many problems and will likely have started again a few times with the Android WebKit batch.

4. If you have them, a batch of unusual browsers like UC, QQ One, Tizen, Puffin, and game consoles. You may want to ignore a few bugs for these rare birds. Don't tell anyone.

5. Finally, the easy browsers: Safari on all iOS devices you own, BlackBerry WebKit, Chrome, Firefox, and Opera Mobile. These browsers usually behave decently and shouldn't cause too many problems, which is why they should go last.

Don't get too hung up on these exact batches, but I hope you understand the principles. Take a few hours to design the batch list; this overhead time will pay itself back many times over when you're in the thick of mobile testing.

Testing Process

Once you get to the actual testing, the following tips and tricks will help you:

1. Use simple URLs. You don't want to type in `192.168.17.49:8080/testsite/default/Default.aspx` twenty times on software keyboards. I use `quirksmode.org/m` as a standard page and add links to whatever I want to test.

2. Make sure you're testing in the right browser. This sounds a bit silly, but if you have three or four browsers on one device, you may accidentally start up the wrong one and think you're testing in Opera Mobile while you actually have Android WebKit open. This has happened to me several times.

3. Be very finicky and precise with testing gesture-based interactions. Make sure that in each browser your gesture is exactly the same. If it's not, you might find differences due not to the browsers but to slight differences in your gestures. In general, it's best to predefine gestures, and make sure their start and end points are visible on the page; for instance, "swipe from the bottom-left corner of element X to the top of the screen."

4. When you're testing responsive designs it may be useful to see the viewport width and height onscreen. Print out `document.documentElement.clientWidth/Height`. Make sure to do it again `onresize` and `onorientationchange`; these events usually fire when the viewport dimensions change.

Overcoming Outdated Reflexes

There are some reflexes from traditional desktop web development that we have to let go of. The most important ones are our distrust of browser detection and our overuse of JavaScript libraries.

Browser Detection

Traditionally, browser detection is a no-go for web developers. If you distinguish between IE and Chrome through their `navigator.userAgent` strings you can expect some pointed questions. Instead, we've learned you should detect the feature you want to use, and make decisions based on the result of that check.

I have been preaching feature detection since 1998 and played my part in convincing web developers of the perils of browser detection, so it took me a while to acknowledge that the situation on mobile is some-

times different. As soon as I started talking to people with years of experience in mobile, they all told me that some sort of browser detection is a necessary evil. The more experienced a mobile web developer is, the more they rely on server-side browser detection because feature detection doesn't help in certain cases. Consider:

1. Suppose you need "Back" functionality on your website. Certain OSs, such as Android and BlackBerry, have a native "Back" button, and inserting your own button would only confuse users — or, worse, confuse the OS's "Back" functionality.

2. Some Android devices claim to support `input type="date"` and such, but don't actually have the native components to fill in a date. BlackBerry 6's default browser supports touch events and tells you so — even if it is running on a device without a touchscreen.

3. A browser might support animations and transitions but have a poor GPU (or none at all) so that everything slows to a crawl, and the user would be better off without them.

In all these cases, the problem is not with the browser's support for CSS and JavaScript but with physical device characteristics or specific OS functionalities. Detecting the presence of a "Back" button is impossible, and in the other examples the feature detection would return a false positive, since the browser only indicates it supports the feature, but not how bad that support is.

If you encounter use cases like this, it might be time to start detecting browsers. This is something you need a bit of experience with. The

point here is not that you should use browser detection for everything, but that there are certain device features that are undetectable in any other way. I'm not saying you should ditch feature detection, but you may encounter situations where browser detection is a necessary addition to your regular feature detection.

If you decide you need a browser or device detection script, don't write your own. Knowledgeable people have already done the work for you. There's a whole ecosystem of device detection services, of which WURFL (smashed.by/wurfl) and DeviceAtlas (smashed.by/atlas) are the best known. Hand it the UA string of a mobile browser, and it will tell you something about the browser's and the device's capabilities. If you're looking for a pure browser detect, without device information, try WhichBrowser (http://whichbrowser.net/).

JavaScript Libraries

The second outdated reflex is to use a JavaScript library for absolutely everything. This is the sad result of the overreliance on libraries that we developed in the 2006–2011 era, to the extent that some web developers can't even write JavaScript anymore.

I've always had reservations about JavaScript libraries, and they were reinforced by a research paper from April 2012 (http://smashed.by/wwwcon). The researchers measured the battery use of an Android phone while loading several websites, including Wikipedia, and experimented with redesigning one function of that website. Wikipedia's accordion script, which uses jQuery, was replaced by a custom-made function, and the measured energy consumption for downloading and rendering the page fell by one-third.

The size of the download isn't even the issue: library vendors are well aware of that problem and have taken steps to make their files as light as possible. The problem is that the entire library has to be executed, draining the battery as feature after feature is initialized — and your page might not even use most of the features.

The solution to this problem is not to relegate JavaScript libraries to the ash-heap of history, but to ask yourself whether you really need one. If you're building a complex interface with lots of functionalities, the answer will likely remain yes. If you just need one basic effect such as a show/hide toggle or simple form validation, it's time to write the entire script by hand. Not only will that sharpen your JavaScript skills, but it will also make your site perform better. Don't worry about browsers: all of them support simple effects well; it's only when you need complex ones that they start to behave erratically and a library becomes a useful addition to your site.

The Mobile Network

Steve Souders did the fundamental research on mobile connections. Read his conclusions at http://smashed.by/mwhb18a

Mobile networks were set up to accommodate devices that have to be reachable only on occasion — when a device makes or receives a call, and when it sends or receives an SMS. Setting up a mobile connection takes some time, and if the mobile connection remains idle for a while, it is closed down in order to save battery life.

This principle also goes for data connections: when the browser requests assets, it takes roughly two seconds to set up a mobile connection, which then closes down after five to twelve seconds of inactivity.

Two seconds might not sound like much, but compounded with the normal latency of a mobile network and web server, it makes for an annoying wait.

Again, there's little you can do about this, except for one thing: if the user needs data anyway, take the opportunity to load as much as you can. This sometimes means making an educated guess as to what data the user is going to need next, and you may occasionally guess wrong, but that's still better than reopening a mobile connection every time the user needs a tiny bit of data. You could decide to store some data, or even things like web fonts, in the browser's localStorage.

Connection Speed

The most serious problem you can encounter on mobile is a slow connection. While desktop connections are generally reliable in the sense that they don't change a lot, mobile connections may vary immensely if a user is on the move. Besides, it's almost impossible to find out anything that's not instantly obsolete about your users' connection speeds. It's essentially an impossible problem to solve.

Instinctively, web developers assume that a 3G or even a 4G connection is slower than a Wi-Fi connection. This may be true most of the time, but not always. Sometimes a user is in a public space with Wi-Fi, but it's slow or unreliable Wi-Fi used by many people simultaneously. It could also be that in one country the 4G network is overused and slow, while another country has a brand-new 4G network that doesn't yet have all that many users — and thus blindingly fast connections.

In such cases, the user's mobile connection might actually be faster. Do not fall into the trap of assuming that a user on 3G has a slow connec-

tion speed, or that a user on Wi-Fi has a fast one. Connection type is not a proxy for connection speed.

Although measuring connection speed is in fact not all that hard, the problem is that the result is worthless. The connection speed may be decent at the moment you measure it, but what if the user is on the move and goes from cell tower to cell tower — or from Wi-Fi to a mobile connection? Or maybe reception is perfect right now, but the user's train is about to enter a connectionless tunnel. Or the user may reach their roaming limit and the connection may suddenly disappear. Although you can detect all that, you can't define a general download policy in such a changing environment, so I advise you not to try.

***

We're nearing the end. You now have a lot of knowledge that will help you become a consummate mobile web professional. There's just one thing lacking: a quick look at the future of the web on mobile.

Chapter 8

The Future Of The Web On Mobile

Chapter 8

The Future Of The Web On Mobile

We've learned a lot, but what's still lacking is a sense of where we're headed. It's clear that web development has changed and will continue to change, due in large part to the advent of the mobile web, mobile browsers, and likely also native apps. So what's next for the web on mobile?

First things first. Traditional websites, in the sense of a user opening a browser, following a link, and loading and using the site, are not going to go away. Although from time to time you'll hear stories of how people are using apps much more than they use websites — and there's likely a kernel of truth in them — smartphone users expect to be able to just visit a website. It is unthinkable that they lose that ability. Thus, mobile browsers will continue to exist, and web developers will continue to be expected to make websites that work well on mobile devices.

HTML5 vs. Native

Back in 2010, the HTML5 vs. native debate was all the rage. People started to understand that for certain classes of mobile apps, HTML5 and native were competitors. When should we use one, when the other? Or would one of them win out over the other?

After a while it became clear that HTML5 and native both have their strengths and weaknesses: native apps offer a superior user experience; while HTML5 apps offer superior adaptability. A victory of one over the other is unlikely, because different use cases require different technologies.

There is no doubt that, as an environment for creating compelling apps, HTML5 is less capable than native. As a result, new features are being proposed that should bring the web more in line with certain aspects of native apps. We'll review a few of them below.

However, even if HTML5 would support all the features of current native apps, it still wouldn't have caught up. Native environments keep evolving as well, and they go faster than the web because they don't have to worry about anything but their own platform. The addition of a feature to iOS is completely independent of what Android is doing or not doing — the two environments simply do not intersect. Conversely, if a new HTML5 feature is proposed it will have consequences in many dozens of browsers, and will have to be discussed thoroughly. That makes the web slower to evolve than native.

As far as I'm concerned, catching up with native across the board is not the point of web technologies. I feel we should concentrate on those

things that the web does better, or could do better, than native. At the end of this chapter we'll encounter a few.

Emulating Native

Although the web will never catch up with native, emulating individual native features can be an excellent idea. A not inconsiderable amount of thinking and engineering is being spent on it. Which native features does it make sense to emulate? The list below is not complete, but it gives you some idea of what we're looking for.

Connectivity and AppCache

Mobile devices can suddenly lose their connections, or never acquire a connection in the first place. That's why it makes sense to store the data or the HTML, CSS, and JavaScript files on the device itself, so that future uses of the site or app aren't impeded by a lack of connection. Data storage is adequately covered by localStorage, but file storage is more of a problem. Originally, appcache was supposed to solve that problem, but it turned out to be so hard to use that a new Service Workers specification was begun instead. We aren't there yet, but the storage problem is in the process of being solved.

According to Jake Archibald, who did most of the research, appcache is a douchebag. It's not completely unusable, but it has so many caveats and edge cases that you have to be very, very careful when using it. Read the lurid story at http://smashed.by/mwhb19. The new Service Workers spec attempts to avoid the appcache mistakes, but at the time of writing it hasn't been implemented in consumer versions of browsers yet. The specification is at http://smashed.by/mwhb20.

Install on Home Screen

A second problem is installation. In the past five years consumers have grown used to installing apps on their devices, and seeing the app icon on their home screen. Mobile browsers support a similar mechanism, but the process is too convoluted.

What we should do is rename the good old "Bookmark" feature to "Install." If a user installs a site, an icon automatically appears on the device's home screen. Tapping the icon will start up the browser and go to the URL in the bookmark. Alternatively, if the website indicates it can be used as an app (for instance, by having a `manifest` attribute on the `<html>` tag) it should be saved to the device and started up when the user taps the icon.

> Chrome for Android has an "Add to Homescreen," function, but only for websites that contain a specific meta tag, and "Bookmark" is still a separate feature. http://smashed.by/mwhb21 has the details. I hope the meta tag restriction will be removed soon and the mechanism sketched in the main text will be implemented.

At the time of writing no attempt is being made to actually implement this, with the partial exception of the Chrome feature mentioned in the sidebar, but I hope that by the time you read this a real solution is in the works.

Device APIs

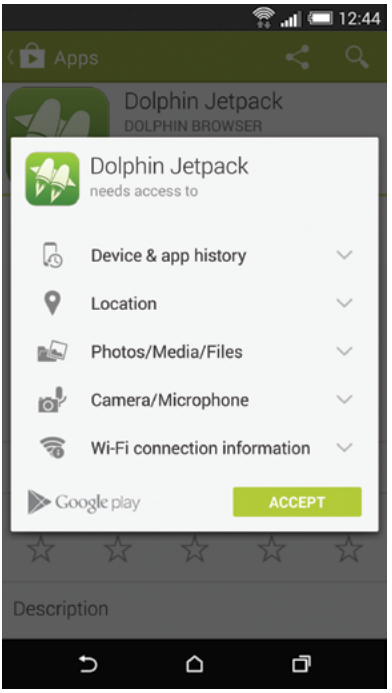
The elephant in the room is device APIs. In order to do something meaningful with devices you should be able to access the address book, sensors, location, SMS capabilities, battery charge indicator and so on.

This is what device APIs do: they offer a simple JavaScript API to access these features, and sometimes change their contents, as in the case of the address book.

Device APIs are a great idea and they are available but, in general, browser makers implement the simpler ones, such as DeviceOrientation, first. Also, in the past few years many platforms have implemented their own APIs, and standardization is slow in coming.

Besides, there are serious security issues. Nobody wants every random website they visit to be able to read their address book and send it off to a malicious server. The user has to somehow grant permission for this sort of device access. Although that's technically possible, presenting this choice in a good user interface is problematic.

So although it's clear that the web needs device APIs to make full use of device capabilities, it will likely take some more time before they're fully supported.



This is the Android model for permissions, and it doesn't work. When you install an app you're asked to give permission for anything the app might try to do. Not only should we have a more granular approach to permissions, but the timing is wrong as well: at installation time the user just wants to get on with it and will accept anything.

It's hard to get an overview of which device APIs are supported where. I found a March 2014 article at http://smashed.by/device-apis that gives some details. It seems that only geolocation, device orientation, media capture, and vibration are supported in more than one browser.

One solution to this lack of support is the PhoneGap project, which was set up precisely to combat lack of device APIs, or their fragmentation. You create a web app using the PhoneGap APIs, submit it to their build system, get back hybrid apps for most platforms, and submit those to the relevant app stores. The system makes sure their APIs hook up with whatever APIs the platform supports.

PhoneGap comes in two flavors: Adobe-owned PhoneGap (http://smashed.by/mwhb22) and open-source Cordova (http://smashed.by/mwhb23). Take a look at both if you need device APIs right now.

Emulating The Web

So far we've discussed native features the web is copying. However, this sort of influence works both ways. Let's take a look at web features that native is, or should be, emulating.

First, URLs. The Applinks service (http://smashed.by/mwhb24) offers ways to deep-link into native iOS, Android, and Windows Phone apps. The process is fairly convoluted, but that shouldn't surprise you. URLs are a quintessential web feature that have no meaning in native. (OK, there's a URL for the App Store or Play page, but that's not the same.)

Still, the people behind Applink think that native apps need this functionality in the long run. It's too early to tell if they're right, but the very fact that native tries to copy this web feature is interesting.

In addition to the universal look-up system implicit in URLs, the web also affords much more adaptability than native, especially when it comes to screen sizes. We web developers know that we can't predict on what size of screen our site or app will be displayed. Besides, CSS gives us the tools to solve this problem. Native developers need a similar solution. Back in 2011, I found an article (http://smashed.by/mwhb25) that called on Android native developers to think more like web developers. I'm not sure what came of this but, again, we see that native developers find they need a few features that we web developers take for granted.

A second way that the web is more flexible than native is just-in-time interactions. Scott Jenson wrote a seminal article on this (http://smashed.by/mwhb26). Essentially, if every shop in a shopping mall offered you a native app for browsing its current discounts, store locations, and other features, few people would bother to download them, since they'll be used only once for one just-in-time interaction. Conversely, if these shops offered a simple mobile website with the same data, people would use them more, since the browser makes it easy to visit a site once and then forget about it. Quite apart from technical features, native apps aren't suited to all situations.

Sharing Apps

It's clear by now that the web and native influence each other in subtle ways, and that is not going to stop any time soon. Still, we haven't yet found a killer feature that highlights the web's strengths.

I'd like to propose one, and again it has to do with superior adaptability. The unique feature of the web is that it works everywhere; that it can adapt to any environment. Any website or web app will work on any device with a browser. On the native side that's unthinkable: an Android app will never work on iOS.

So let's allow users to share web apps from phone to phone. I have an Android phone with a nice app; I show it to you and you want it, too, but you are on Firefox OS. No problem: I open a peer-to-peer connection (Bluetooth, NFC, whatever), send over the app, and you can use it straight away.

The really annoying bit is that this is not some pie-in-the-sky utopian idea, but something I actually did back in 2009, when I worked at Vodafone. I created a lot of test apps for the now obsolete W3C Widgets web app platform, which ran on Symbian. When I got my first Windows Mobile phone I went through the specifications, saw it supported W3C Widgets, and decided to test it. So I opened a Bluetooth connection from Symbian to Windows Mobile, sent over a test widget, opened it on Windows Mobile, and lo and behold: it worked. An app written for Symbian ran on Windows Mobile. Ever since I've been convinced that app sharing is the future of the mobile web.

Meanwhile, unfortunately, this idea has gone out of fashion, partly because there are serious security issues akin to the ones with device APIs — you don't want a random app to send off your address book to an unknown server. Still, I assume that this idea will make a comeback because the concept is so simple any consumer will understand it, and because it highlights the unique capabilities of the web that native apps cannot emulate.

***

We have come to the end of this book. My hope is that you've learned something about the mobile web, and how it sometimes differs fundamentally from the desktop web.

Still, don't take everything this book says as the gospel truth. You should see it not as the end of your journey into the fascinating, and also confusing, mobile world, but as a travel guide for your first steps. I'm certain that mobile, and new devices in general, are going to change the web beyond our wildest dreams.

So let's move on. It's going to be quite a journey.

Index