O'REILLY®

High Performance Mobile Web

BEST PRACTICES FOR OPTIMIZING MOBILE WEB APPS

Early Release
RAW & UNEDITED

Maximiliano Firtman

High Performance Mobile Web

Maximiliano Firtman

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'REILLY®

High Performance Mobile Web

by Maximiliano Firtman

Copyright © 2010 Maximiliano Firtman. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Editor: Brian Anderson
Production Editor: FIX ME!
Copyeditor: FIX ME!
Proofreader: FIX ME!
Indexer: FIX ME!
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

: First Edition

Revision History for the First Edition:

2015-07-10: Early release revision 1

See *http://oreilly.com/catalog/errata.csp?isbn=9781491912522* for release details.

ISBN: 978-1-491-91252-2

[?]

Table of Contents

CHAPTER 1

The Mobile Web World

The first question you ask yourself before getting this book into your hands is: why are we talking about performance on the mobile web as a topic? isn't the mobile web the same web as the one we already know?

The answer is not so simple. Yes, it's the same web; but it's being accessed from a very different context, including browsers, devices, screens and networks. Those differences have a big role in performance -or in the lack of it- and that's why we need to pay special attention to the web on mobile devices.

Fortunately, following the idea originally stated by Luke Wroblewski of *Mobile First*, if you first apply performance techniques for mobile devices, your website will also perform well and faster on other kind of devices including desktop browsers and TVs.

Mobile Performance First

If you have a multi-device web solution, starting to optimize the performance of the mobile web will also help other devices, such as desktop browsers. While most techniques over this book can be applied on both classic and mobile web, some of them are specific to mobile device's problems and therefore easier if we start with them.

Don't get me wrong, I'm not pushing the idea of having a separate web. At the end, we are still talking about the same content and the same services but from a different devices. In this chapter we'll cover the differences from the desktop web, which we will call from now on *the classic web*, that is the web from the main browsers on typically desktop-based OSs, such as Windows, Mac OS X or Linux.

Therefore we'll define the Mobile Web, as web content being accessed from a feature phone, smart phone, tablet or wearable device. I know there are some hybrid devices that can fit into the classic web definition, but I think we all know where the line is.

Before getting into the performance side of the mobile web, we need to approach some definitions and clarifications in relation with the differences between classic web platforms -such as on desktop- and web platforms running on mobile devices.

Form factors

Several form factors are available on the market, but if we focus just on mobile devices with web platforms we can divide them into:

- TV-based devices, including: Smart TVs, Game consoles (such as Xbox or PlayStation) and Set top boxes (such as Chromecast, Amazon Fire TV or Roku Player)
- Desktop devices, including laptops using OSs such as Windows, Mac OS X, Linux or Chrome OS
- Tablets
- Phablets, phones with screens bigger than 5.5”
- Smartphones, phones using a *big OS* such as Android, iOS, Windows or BlackBerry
- Social devices, cheap phones with web access using operating systems as Firefox OS, Nokia Asha or feature phones
- Smartwatches

In this book we’ll focus on smartphones, phablets, social phones and tablets but some techniques and tools might be useful also for other form factors.

Mobile Hardware

In the future it might be a point when mobile devices will be more powerful than desktop and laptops (mostly because the technology is going in that direction), but we are not there yet. Mobile devices have less RAM available and less powerful CPUs.

I know you might think this is not true, in an age of quad-core mobile devices but the reality is that the average mobile device out there is not the most expensive device.

> If you are reading this book, you probably don’t have an average phone in terms of hardware and network access. When thinking about performance, always think about average users and test on those devices.

There are dozens of differences between mobile devices and classic desktop devices, such as screen resolutions, screen densities and hardware sensors but the ones that might affect web content’s performance are:

- CPU, where the parsing, rendering and execution happens
- Memory, where the DOM tree, image buffers and decompression data are stored
- GPU, where -when available- some rendering happens (usually known as hardware accelerated)
- GPU memory, where -when GPU available, some image buffers and layers are stored

We can roughly say that in terms of CPU, average mobile devices are 5 times slower than an average desktop device acquired at the same time. In terms of RAM memory, the difference is around 3 times smaller.

Mobile phones are still behind desktop hardware

You can say that because the screen is also smaller having less power and memory won't make a big difference, but if you think about this twice that's not really true.

The desktop computer I'm using to write this book (a high end laptop) has 8Gb of RAM, 1.5Gb of CGU memory and 1.3 millions pixels on the screen. My current phone -a high end Android phone bought at the same time as my laptop- has 2Gb of RAM and 2 millions pixels on the screen. In terms of CPUs, they have different architectures (Intel vs ARM) so it's difficult to compare them by their specs, but based on CPU Boss Comparison my laptop's CPU is 2.7 times faster than my phone's CPU.

Therefore, just in my own example, my phone has 53% more pixels to draw on the screen with a quarter of the RAM and roughly a third of CPU power when compared to my laptop acquired at the same time. And remember, I'm don't have an average device. That only means performance problems.

Mobile Networks

> We have 4G now! We don't have performance problems on the network anymore.
>
> — a random web developer

Mobile devices connect to the network in different ways, usually using a WLAN (Wi-Fi) connection or a cellular connection.

Wi-Fi accesses can be reliable (such as a home or office connection) or unreliable (such as on a coffee store, a plane or a bus -which usually goes through a cellular connection).

Cellular connections are fare less reliable compared to Wi-Fi access because of the nature of the system. We are connecting wireless through a radio to a remote wired connection (in a cell's tower up to a couple of miles from us). The connection changes cells, availability and connection type frequently. More frequently if we are on the move.

Cellular connections today are usually known with popular names as 2G, 3G and 4G. Inside those names, several different connection types can be used, such as:

- GPRS (2G)
- EDGE (2G, 2.5G or 3G)
- UMTS (3G)
- WCDMA (3G)
- HSPA (3G)
- EVDO (3G)
- HSPA+ (3G or 4G)
- Mobile WiMax (4G)
- LTE (4G)

Distribution

While you might think that 4G is now everywhere, that's because you are reading this book are you are probably a techie user. The reality out there is far from being similar.

At the end of 2014, according to GSMA Intelligence 64% of the world is connected through 2G, 30% through 3G and only 6% through 4G.

Values changes when talking about developed countries, including US, but it's still 20% of the users on 2G, 55% of the users on 3G and just 25% on 4G.

4G Americas has also a chart of the 4G penetration as of mid 2014, being 35% in United States, 8% on Western Europe and 3% on Asia.

At the end of 2014, 64% of the users worldwide were using 2G connections

The other important data here is that even when you have a 4G device and you have a 4G carrier, 70% of the time you might not be using that connection but downgrading to 3G or 2G. Open Signal has great deep information on this problem and they state that US 4G users are not using 4G 32% of the time while in Germany it can be up to 50%.

All this data means that we will live with 2G and 3G connections for a while and we need to take special attention to this when we are providing mobile services.

Bandwidth

The first thing that comes to our minds when talking about 3G or 4G is bandwidth. And it's true: better the connection means higher bandwidth.

The bandwidth available in average in each type of connection is:

Table 1-1. Mobile network speeds in Megabits/second (Mbps)

network	minimum	maximum	average[a]
2g (edge)	0.1	0.4	0.1
3g	0.5	5	1.8
4g	1	50	3.4 (HSPA+), 7.5 (LTE)
Wi-Fi[b]	1	100	6.1

[a] data by Open Signal State of LTE Report

[b] measured by Open Signal on mobile apps

However, the question for us is: how important is the bandwidth when we are delivering mobile web content? More bandwidth is usually better but the difference will not be huge because we are transferring just a bunch of small files. If you are doing video streaming then bandwidth will be more important.

Latency

And here comes the biggest problem on mobile networks: the *wireless network latency*. The latency is the time it takes for the mobile device to send a data packet to the server. Usually we measure the Round Trip Time (RTT) latency, that is the time it takes to get the first byte from the server after making a request.

Latencies can be up to 1 second on 2G networks, that means the browser will wait 1 second per request before starting to receive any actual data. Even on 4G (that remember, it's just a small percentage of users worldwide) the RTT can be up to 180 milliseconds. Just to make a comparison with your home wired connection, a DSL in United States has a RTT latency of 20 to 45 seconds.

The next table shows you the latencies we can found today in mobile networks.

Table 1-2. Mobile network speeds in milliseconds

network	minimum	maximum	average	compare to average home]
2g (edge)	300	1000		+20x
3g	150	450		+9x
4g	100	180		+4x

Latency RTT is a big problem for the mobile web, because a typical mobile website or web app is a group of small files that travels through the network as several request, each one with one RTT overhead.

If you are wondering why do we have the RTT latency, it's caused by the network architecture involving:

1. The travel time from our phone to the cell tower (wireless)
2. The travel time from the cell tower to the carrier gateway (wired or wireless)
3. The travel time from the carrier gateway to the internet (mixed)
4. The same travel time (1 to 3) in reverse order

When you are connecting to the web using a home or office connection you are skipping travel times 1 and 2 from the previous list.

Radio state

The final characteristic of a mobile network is radio state changes. Your mobile device has a radio, that is the one used to send and receive data from the cell tower. That radio consumes battery, so the mobile OS usually tries to save its usage to reduce battery consumption.

That means that if no app is using the network, the mobile device will switch the radio from *active state* to *idle state*. As a user, you don't know the current radio's state. When any app or website is trying to gather some data again from the network, the device will restart the radio, and that usually involves time. On 3G connections, the time can be up to 2.5 seconds, while on 4G connections less than 100 milliseconds.

Therefore, if the radio was in idle state, the first RTT latency will also have an overhead.

You might be thinking: I'm not a network operator, what can I do from a web point of view about this? Later in this book we'll see techniques and best practices to improve performance and perception thanks to having an understanding of these network problems.

Mobile Operating Systems

For the purpose of this book we'll focus our attention into the main mobile operating systems based on current market share. Most of the explanations and techniques will also be useful for other OSs and we'll make some comments when something might be different.

Main operating systems:

- iOS
- Android
- Windows

On Windows we'll focus on the versions available at phones (formerly known as Windows Phone) with some comments on Windows for tablets.

Fire OS is an operating system created by Amazon which powers the Kindle Fire series and the Fire Phone. Fire OS is a UI layer on top of Android, so from an OS' perspective we'll be considering it as Android. Nokia had a similar approach for one year with Nokia X, an Android-based platform that was canceled by Microsoft in mid-2014. Both projects are using different web runtimes and web browsers compared to the original standard Android flavors meaning different techniques to test and debug for performance.

Besides the main operating systems, on the mobile space today we can also find devices using:

- BlackBerry 10
- Firefox OS
- Tizen
- Sailfish

There are also obsolete operating system that we won't cover in this book for being less than 0.5% of the market at the time of this writing and because they don't have any future, including:

- Symbian
- MeeGo
- webOS
- Nokia Series 40

While most techniques in this book will be also useful on desktop devices, we will not focus here on Windows for desktop, Mac OS X, Linux or Chrome OS.

Engines

Before getting into the analysis of the current state of the mobile web, let's make a quick review of web rendering engines and execution engines that will be useful to understand some references in the rest of this book.

Rendering engines

The rendering engine (also known as layout engine) is the code that will be downloading, parsing and rendering your HTML and CSS code on the screen, as well as other related content, such as SVG or images.

With regard to mobile devices, the king of rendering engines was WebKit for years, empowering browsers on iOS, Android, BlackBerry, Symbian and many others.

Today, in the mobile space we can find the following rendering engines available:

- WebKit
- Blink
- Trident
- Gecko
- Presto

WebKit is an open source project created in 2001 by Apple based on KHTML (other engine available for Linux-based browsers created in 1998) to support the Safari browser. The component related to layout and rendering is known as WebCore. Many companies have been using WebKit on the mobile space for years including BlackBerry, Nokia, Google, Samsung and Palm, besides Safari on iOS. It's today the most used rendering engine in the mobile space.

Most of the times, when we are talking about WebKit, we are usually talking about WebCore, a part of WebKit that is responsible for parsing and rendering HTML and CSS

The great thing about WebKit is that most web platforms in the mobile world are using it. This means that even on very different mobile devices we can expect very similar web rendering with simple markup and styles, which is good news for developers. However, it isn't perfect —as we'll see in later chapters many differences do exist between WebKit implementations, as well as in HTML5 compatibility.

Google Chrome was using WebKit until 2013, when it decided to fork it and continue it in a separate path with the name *Blink*. Today, in terms of performance Blink and

WebKit are different even sharing sources. Blink empowers today Google Chrome and Opera.

WebKit2 is a new project created from the ground up supporting a new internal architecture that splits the web content process from the application UI process. At the time of this writing the only mobile browser who has supported this new version was the Nokia Browser for MeeGo, an obsolete platform today.

Trident is the proprietary engine inside Internet Explorer while Gecko is an open source engine managed by the Mozilla Foundation. Both of them didn't gain so much track in the mobile space yet but they have enough market share to pay attention to them. From Windows 10, a new fork from Trident has born on Microsoft Edge.

Finally, Opera was using Presto -their own proprietary engine- for years on their mobile browsers. Since Opera 12, they have switched to Blink removing future updates of Presto. However, even today there are still users browsing the web with older versions of Opera using this engine.

Execution engines

When you have JavaScript code, the execution engine comes in action. It's the runtime that interprets your code, manage the memory for your variables and objects and interact with the platform behind -usually, a browser-.

We need to separate the life of JavaScript execution engines in two eras: classic and JIT (Just in time) compilers. Classic engines were interpreting JavaScript while executing, while JIT engines pre-compiles the JavaScript code into native code before executing it with a big improvement in terms of performance.

The first of the new engines was *Carakan*, created by Opera for its desktop browser; this was followed by *V8*, an open-source JavaScript engine developed by Google that is currently used in Google Chrome and other projects such as *node.js*. Other modern JavaScript engines include *JägerMonkey* (Mozilla Foundation), *IonMonkey* (Mozilla Foundation, since 2013), *Nitro* (also known as SquirrelFish), and *Chakra* (Internet Explorer).

Execution engines with Just in Time (JIT) compilers can run JavaScript code between 3 and 5 times faster on the same device.

Old engines with no pre-compilation features are JavaScriptCore -part of the WebKit project- and SpiderMonkey -originally on Netscape, later on Firefox-.

Web Platforms

A web platform is the scope where our web content is being executed and parsed. We can start categorizing the mobile web platforms today into:

- Web Browsers
- Web Apps
- Web View-based content
- e-books

E-books is out of the scope of this book, but let's say for now that latest version of epub (the standard format for publishing ebooks) and mobi (the Amazon propietary format) are using HTML5-based content. Therefore, interactivity comes in the form of JavaScript running inside the ebook reader.

While I'm pretty sure you already know what a web browser is, you might have some doubts about the exact definition of the other platforms. Let's so start talking about them before getting into the world of web browsers on mobile devices.

Web Apps

While the definition is not 100% written in stone, at least from this book's perspective, a web app is a web content that was installed in the Applications menu or the Home Screen of a device and it has a chrome-less user interface in the operating system.

You can think on a web app as a website on steroids; from a user's perspective it looks like an installed app, but from ours perspective we are talking about web content in HTML5, CSS and JavaScript.

Web apps synonymous
Web apps are sometimes known as *home screen webapps* or *full screen webapps*.

From a performance's point of view, a web app includes:

- A server that is hosting the files
- A possible local resource installation
- A declaration of meta data for the installation process

- A web rendering engine and a JavaScript execution engine loading the content, with or without the context of a browser process
- There is no compilation or package process
- Web apps can be installed from a browser or from an app store

While on some OSs a web app runs on exactly the same engine as a web site, on some other OSs the engine might defer meaning different performance measurements and techniques.

At the time of this writing, the web app platforms available are:

- iOS Home Screen Web apps
- Chrome Home Screen Web apps
- Firefox Open Web apps

There were some other platforms supporting web apps that today are considered obsolete, such as Series 40 Web apps or MeeGo Web apps both from Nokia, before Microsoft's acquisition.

Native vs. web site vs. web app vs. hybrid

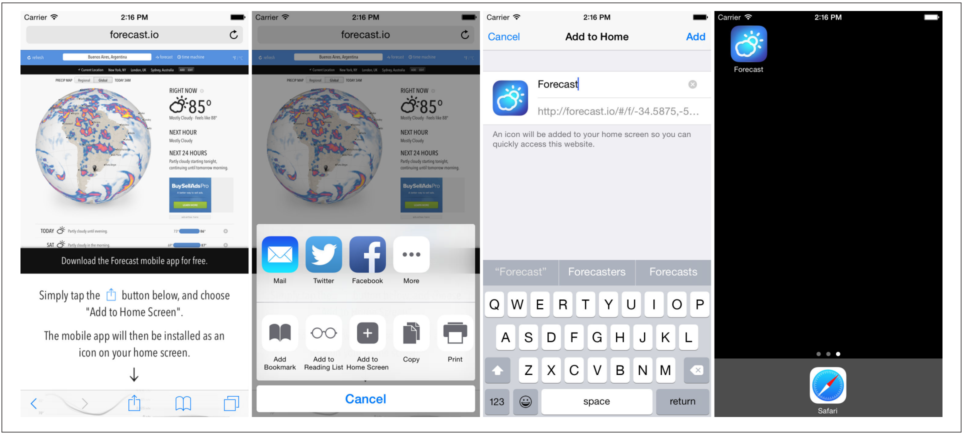
In this book I'm not getting into the discussion of which approach is the best one per case. In this book we will cover techniques applicable for all the options but no the native. The next table shows you the features unique to its kind:

Table 1-3. Mobile approaches

approach	distribution	usage	development	full native access
web site	browser	url, social sharing & search engine	html5 & web	no
web app	browser	icon	html5 & web	no
hybrid	app store	icon, custom uri (web & other apps)	html5 & web	yes (throught native code)
native	app store	icon, custom uri (web & other apps)	native code	yes

iOS Home Screen Web apps

On iOS -the OS behind iPhones, iPod touches and iPads- Safari has an option to *Add to the Home Screen* any website. By default, when the option is activated, Safari adds a shortcut to the website at the home screen as we can see in Figure 1-1.


Download the Forecast mobile app for free.

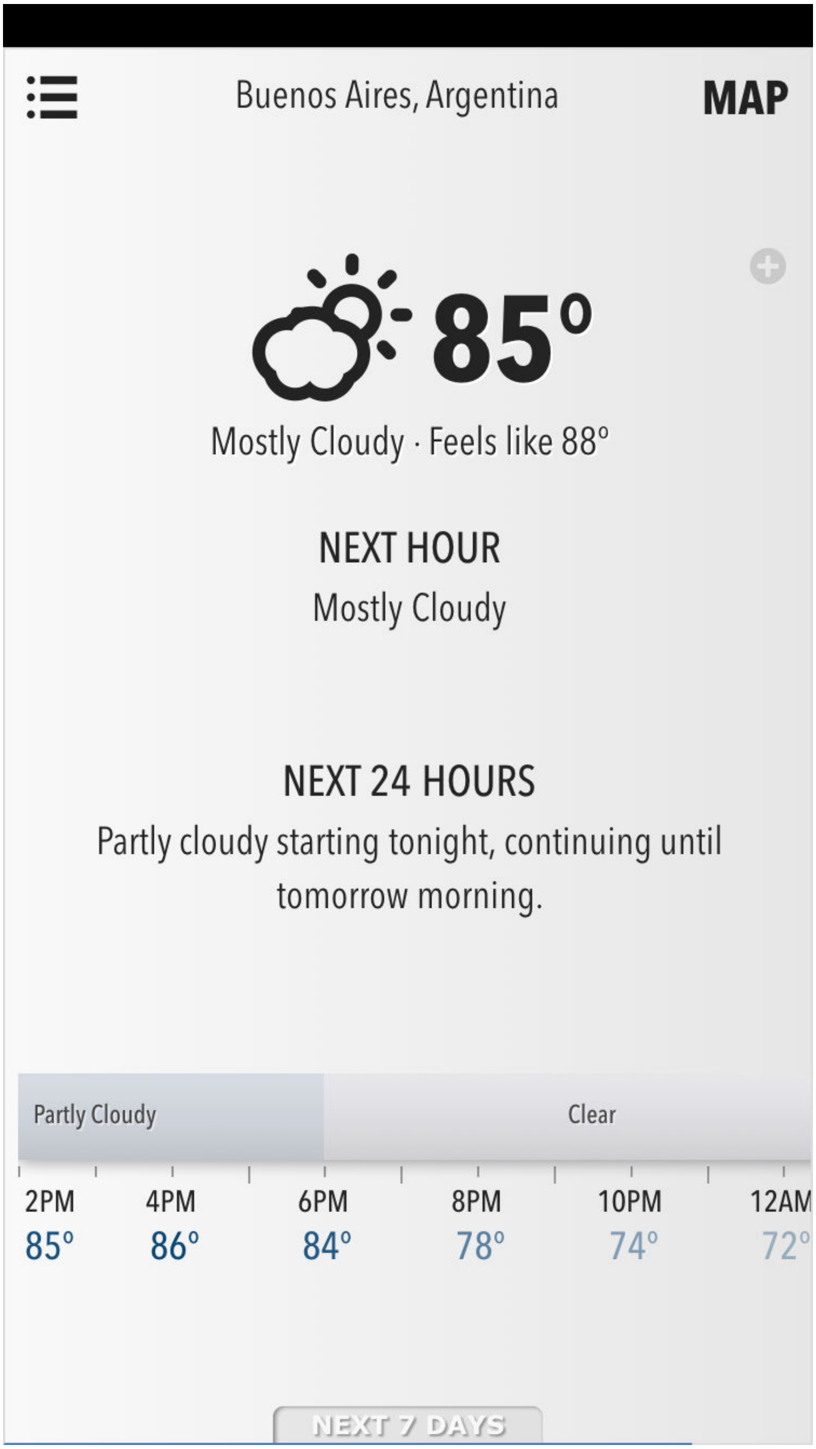
Simply tap the button below, and choose "Add to Home Screen".

The mobile app will then be installed as an icon on your home screen.

An icon will be added to your home screen so you can quickly access this website.

Figure 1-1. On Safari we can add a website to the Home Screen to get an icon as an Application. Here forecast.io suggesting us the process.

If the website has the proper declaration through `<meta>` tags, when the user adds the website to the Home Screen, the icon becomes a Web app. A Web app runs in full screen mode, it appears as a full app from an OS' perspective and it's not running under Safari as we can see in Figure 1-2.



Buenos Aires, Argentina

MAP

85°

Mostly Cloudy · Feels like 88°

NEXT HOUR

Mostly Cloudy

NEXT 24 HOURS

Partly cloudy starting tonight, continuing until tomorrow morning.

Partly Cloudy

Clear

2PM	4PM	6PM	8PM	10PM	12AM
85°	86°	84°	78°	74°	72°

NEXT 7 DAYS

Figure 1-2. When a web app has the right meta data, it runs in full s[illegible]e, out

side of the scope of Safari. From an OS' point of view, it's a separate app from Safari.

If you have never seen a web app on iOS, you can try these examples from Safari iOS: Financial Times, Weight Tracker or Forecast

Web apps on iOS run under a process known as *Web*. This process is using the same WebCore and Nitro engine as safari but it also means it can add its own set of features and bugs.

iOS web apps' didn't have the JIT Nitro engine for a couple of versions when this faster engine appeared on Safari. This absence lead to several critics on the press about Apple not pushing web technologies. From version iOS 5.0 the Home Screen Webapps are using the much faster engine instead of the older JavaScriptCode.

On iOS, home screen web apps are usually target of several bugs when a new version appears. If you are creating these kind of solutions have in mind that you should re-test your app after each iOS release.

Chrome Home Screen Web apps

Google Chrome for Android since version 33 also supports Home Screen Web apps using the same mechanism: the user has to use the menu to pick the option and a new icon will appear on the Home Screen as we can see in Figure 1-3.

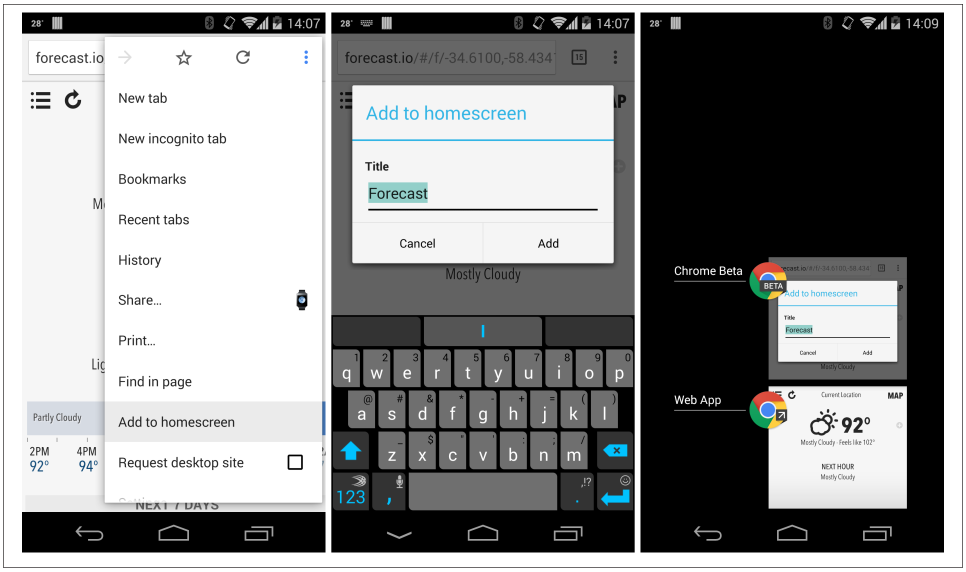


Figure 1-3. Chrome on Android has the ability to install a Web app through the drop down menu. Here we can see forecast.io installation step by step.

As in iOS, the Web App appear from an OS point of view as a different document and in terms of performance and engines, it is using the Chrome you have installed on your phone. That means that if you update Chrome from the Play Store, the web app's engine will also be updated.

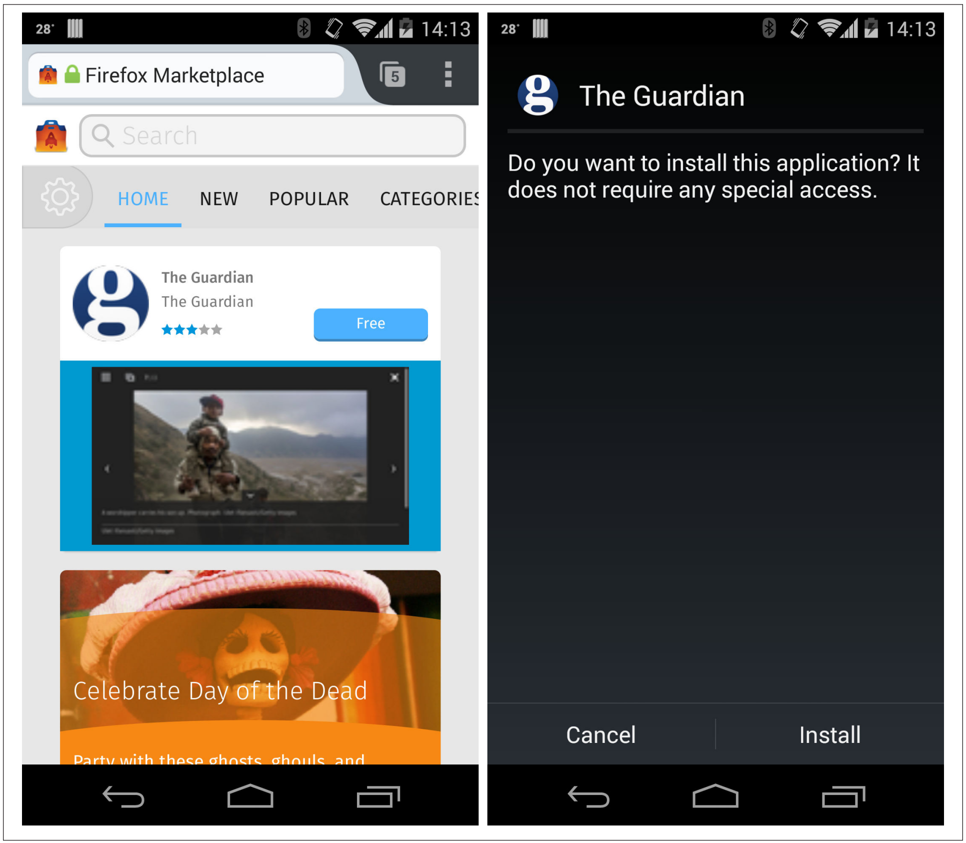
Remember that a website is not a web app after adding it to the home screen by default. The website has to declare a manifest file for Chrome to consider it as a web app. Without the manifest, the icon on the home screen will just open Chrome as normal website.

If you want to learn more about Chrome Home Screen Web apps visit Chrome Developer's guide.

While sharing engine and code from Android 4.4, Home Screen Web apps and WebViews have different compatibility and performance while running on the same device

Firefox Open Web apps

Firefox has created an open platform for web apps, but only their products are currently using it, including Firefox for Android and Firefox OS. Firefox OS is an operating system available for mobile devices based on a low level Android-based core and a high level Firefox UI.



Do you want to install this application? It does not require any special access.

Figure 1-4. On Firefox OS, or Firefox on Android we can install web apps from a website or from the Firefox Marketplace

Open Web apps can be installed from any website -with a proper manifest file and a JavaScript API- or can be installed from the Firefox Marketplace, the official store for the platform. In the latter case, Mozilla will host the files of the web app for us.

Every app inside Firefox OS -including system apps- is HTML5-based running under the Firefox engine behind the current version of Firefox OS. On Android, if the user has installed Firefox manually, then she can install web apps to the home screen with a similar behavior as in Chrome for Android. Once installed and opened, it runs under the Firefox engine but looks like a chrome-less full app from a user's perspective.

Summary of Web Apps

Web apps are a way to install apps without the need of a native compilation and a store distribution, based 100% on web content. In terms of engines available, the following table will bring us information.

Table 1-4. Web apps' engines

OS	devices	versions	rendering	execution	process
iOS	all	2.0-4.3	WebKit	JavaScriptCore	Web.app
iOS	all	5.0+	WebKit	Nitro	Web.app
Android with Chrome	all	4.0+	Blink	V8	Chrome
Android with Firefox	all	2.3+	Gecko	SpiderMonkey	Firefox
Firefox OS	all	all	Gecko	SpiderMonkey	Firefox

Remember, when we are talking about web apps we are not packaging the contents in a ZIP or similar container such as APKs (Android Packages) or IPAs (iPhone Applications). We are using just a normal web server to host the files, and those files will run in a full-screen environment.

Web Views

A web view is a native control available on most platforms (similar to Buttons, Labels and Text Inputs) that allows a native app to include and run web content. The web view can use the entire screen or just use a small portion of it.

While most of the techniques in this book will apply to web view-based content, on Chapter XX we'll focus on these platforms. We are focusing on the web view because while sometimes sharing the same engine as the default browser on that device, performance techniques will vary.

It can be used for different purposes, including:

- Show rich content
- Execute JavaScript code
- Show animations or ads
- Show an in-app browser
- Create a pseudo-browser
- Create the entire user interface and logic with web content (usually known as a *native web app* or a *hybrid app*), such as an app using Apache Cordova or PhoneGap.
- Create a native shell for an online web app

While working with web view-based solutions, we need to understand that the same code on different devices might run on different engines. Even on the same device, after an OS update the engines will be updated meaning that our code is not guaranteed to run in exactly the same way. If we follow good practices we will reduce the risk of incompatibilities in the future.

Native web apps

On some platforms, we have an official platform to create native apps from web content without creating a native project and adding a web view. These platforms are using the web view inside.

Some platforms that will use the web view on the background are:

- Apache Cordova (also known as PhoneGap), including the PhoneGap Build service from Adobe
- JavaScript Windows Apps
- BlackBerry WebWorks
- Amazon WebApps for Fire OS and Android
- Tizen Apps

Compared to web apps covered before, a native web app or hybrid involves:

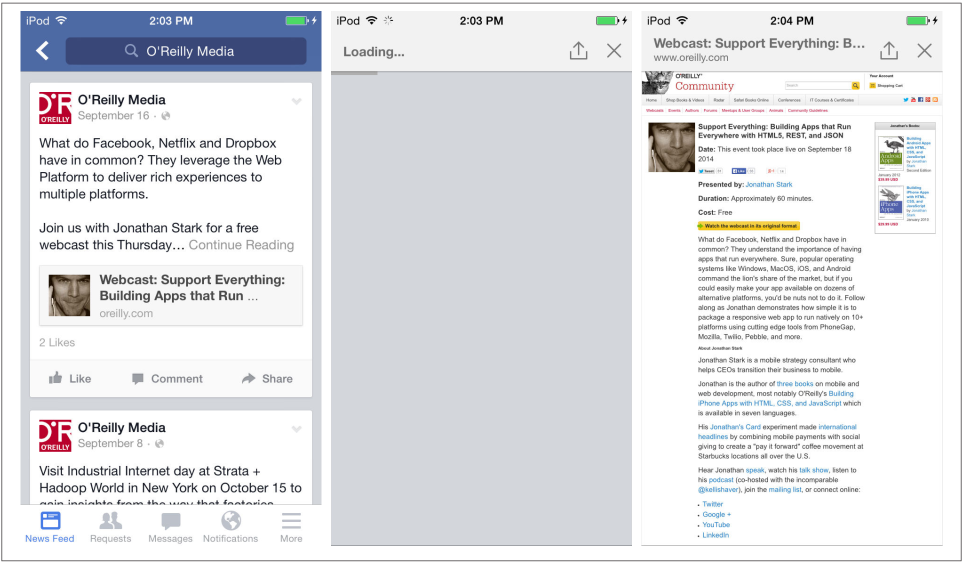
- Packaging the web content into a zip or container
- Creating a manifest declaration
- Sign the package
- Distribute the app as a native app on app stores

We shouldn't confuse a native web app platform running web content on the fly on the device with cross-platform solutions that might start with web content but will be pre-compiled into native code before publishing them. Therefore, no Web View is involved in these cases. Titanium Appcelerator is one example of these tools; you write JavaScript but it will be compiled into native code.

In native web apps we are not hosting the files on a web server but inside the app's package. This will be a big difference when talking about performance.

In-app browser

Some native applications have the need of showing you a web site but they don't want you to get out of their app. This happens mostly on iOS where you don't have a back button to quickly go back to the app after opening the browser. Great examples here are the Facebook, Twitter and Flipboard.



O'Reilly Media
September 16 ·

What do Facebook, Netflix and Dropbox have in common? They leverage the Web Platform to deliver rich experiences to multiple platforms.

Join us with Jonathan Stark for a free webcast this Thursday… Continue Reading

Webcast: Support Everything: Building Apps that Run …
oreilly.com

2 Likes

Like Comment Share

O'Reilly Media
September 8 ·

Visit Industrial Internet day at Strata + Hadoop World in New York on October 15 to

Support Everything: Building Apps that Run Everywhere with HTML5, REST, and JSON

Date: This event took place live on September 18 2014

Tweet 31 Like 33 +1 14

Presented by: Jonathan Stark

Duration: Approximately 60 minutes.

Cost: Free

Watch the webcast in its original format

What do Facebook, Netflix and Dropbox have in common? They understand the importance of having apps that run everywhere. Sure, popular operating systems like Windows, MacOS, iOS, and Android command the lion's share of the market, but if you could easily make your app available on dozens of alternative platforms, you'd be nuts not to do it. Follow along as Jonathan demonstrates how simple it is to package a responsive web app to run natively on 10+ platforms using cutting edge tools from PhoneGap, Mozilla, Twilio, Pebble, and more.

About Jonathan Stark

Jonathan Stark is a mobile strategy consultant who helps CEOs transition their business to mobile.

Jonathan is the author of three books on mobile and web development, most notably O'Reilly's Building iPhone Apps with HTML, CSS, and JavaScript which is available in seven languages.

His Jonathan's Card experiment made international headlines by combining mobile payments with social giving to create a "pay it forward" coffee movement at Starbucks locations all over the U.S.

Hear Jonathan speak, watch his talk show, listen to his podcast (co-hosted with the incomparable @kellishaver), join the mailing list, or connect online:

- Twitter
- Google +
- YouTube
- LinkedIn

Building Android Apps with HTML, CSS, and JavaScript by Jonathan Stark Second Edition January 2012 $39.99 USD

Building iPhone Apps with HTML, CSS, and JavaScript by Jonathan Stark January 2010 $29.99 USD

Figure 1-5. Some iOS apps include an In-app browser mechanism, such as Facebook that allows you to browse a friends' suggested website without opening Safari

On Android and Windows Phone because of the back button on the screen, the user can quickly return to the previous app after a linked website.

According to the Mobile Overview Report (MOVR) from ScientiaMobile, during 2014 5% of the web traffic coming from an iOS device was received from inside the Facebook app.

When you see an app with an ad on iOS or Android there is a good chance that it's using a web view because both iAd from Apple and AdSense from Google are using this control for in-app advertisements.

Pseudo-browser

I'm pretty sure you've never heard about *pseudo-browsers*. It's really a new category that I've created a couple of years ago. A pseudo-browser is a native application marketed as a web browser that, instead of providing its own rendering and execution engines, uses the native web view.

From a user's point of view, it's a browser. From a developer's point of view, it's just the web view with a particular UI. Therefore, we have the same rendering engine as in the preinstalled browser, but with a different UI. These pseudo-browsers are mostly available for iOS and Android, and they offer the same service as the native browser but with different services on top of them.

On iOS the Web view is being upgraded with the operating system. There is no way to update or change it as a separate component. Because of an App Store's rule we can't provide our own engine inside the iPhone App either as in other operating systems.

This can lead to some philosophical questions about what a browser is. From a developer's perspective, it's important to understand that pseudo-browsers are not adding fragmentation—they're not new engines, but simply the web view from the operating system.

Chrome on iOS is the best example of a pseudo-browser; because Apple has a restriction on using third-party engines, Chrome is using the iOS Web View instead of a Blink-based engine.

xxx

The main difference with an In-App browser is that in that case, the app is not marketed as as a browser, you can't say that Facebook for example is a web browser, even having an In-App browser now on iOS and Android.

The Web View on iOS

The Web View available on iOS is WebKit based and while some people will consider the web view as an embedded Safari, the reality is different.

First, on iOS since version 8.0 we have two Web Views available: the old one (known as *UIWebView*) and the new one (known as *WKWebView*). While we will talk more about this later in this book, from our point of view it's important to understand that performance on both controls will be different.

Apple will discontinue the old version with time but for now, let's say that if you are using a web view and you didn't explicitly changed the web view you are still using the old one.

To clarify the engines behind iOS, let's see the following table

Table 1-5. iOS web engines

type	versions	rendering	execution
Safari	1.0-4.2	WebKit	JavaScriptCore
Safari	4.3+	WebKit	Nitro JIT
Web app	2.0-4.3	WebKit	JavaScriptCore
Web app	5.0+	WebKit	Nitro JIT
UIWebView	all	WebKit	JavaScriptCore
WKWebView	8.0+	WebKit	Nitro JIT

On iOS the difference between using JavaScriptCore and Nitro with the JIT compiler is in the magnitude of 3. That is, on the same device, a JavaScript code running on UIWebView will be at least 3 times slower as the same code running on WKWebView or Safari. However, we will see later JavaScript execution time is just one part of the total time a page takes to load so it doesn't mean your website loads 3 times slower.

The Web View on Android

For years the Web View on Android was the cause of many companies leaving the usage of HTML5 for creating native apps. The Web View was extremely slow on Android 2.x, getting better on 4.0 but not good enough until 4.4.

Until Android 4.3, the Web View was based on WebKit and from 4.0 to 4.3 it was pretty much the same version without any changes or updates in performance or compatibility.

Because of this problem, some Android manufacturers, including Samsung and HTC have decided to replace the default Web View with a modern version of WebKit or Chromium -the open source version of Chrome with Blink and V8, much faster and compatible with modern HTML5 features. The problem is that on each device and version of the OS the version of WebKit or Chromium might be different, as well as the features and abilities enabled.

Amazon for Kindle Fires has kept the original Android Web View but also added a second Web View, known as *Amazon Web View* based on Chromium 18 from the Kindle Fire 2nd generation. Finally, on the Fire Phone, Amazon has decided to just replace the default web view on Android 4.2 (the version they are using) with their Amazon Web View.

From Android 4.4 (KitKat), Google has decided to replace the Web View on Android with Chromium to keep everything under one umbrella again, starting with Chromium 30 (equivalent to Chrome 30) and following with version 33 on the small 4.4.1 update.

Even if a user has installed Chrome -or Chrome came pre-installed- a user on Android 4.0-4.3 will execute Web view-based code on WebKit and a non-JIT JavaScript engine by default.

Since Android 5.0 the Web View is based on Chromium 39 and it has become -for the first time- updatable from the Google Play Store, meaning that the user can install new versions of the Chromium-based Web View without waiting a next version of the OS. Therefore now Web view-based solutions can keep up-to-date with Chrome new features.

The WebView world on Android looks today like the next table:

Table 1-6. Android web view engines

devices	versions	rendering	execution
All[a]	1.0-4.3	WebKit	JavaScriptCore
All	4.4	Chromium 30	V8 JIT
All	4.4.1	Chromium 33	V8 JIT
All	5.0+	Chromium 37 (updatable via store)	V8 JIT
Samsung	4.1-4.3 (some devices, such as S3 & S4)	WebKit	V8 JIT
Samsung	4.1-4.3	Chromium (Blink)	V8 JIT
HTC	4.1-4.3	WebKit	V8 JIT
Amazon Kindle Fire (using alternative webview)	4.0+	Chromium (Blink)	V8 JIT
Amazon Fire Phone	4.2+	Chromium (Blink)	V8 JIT

[a] Not including Samsung, HTC and Amazon devices with custom web views.

Intel has created an open source project -based on Chromium- called Crosswalk that offers us an injectable WebView for Android 4.0+, supporting latest versions of Chrome. If we use it we will be normalizing all the web views on Android but delivering a bigger Android App file.

The Web View on Windows

The web view on Windows (formerly also known as Windows Phone), such as on Microsoft Lumia devices, is far more simple than on Android or iOS. The Web View is using Trident and Chakra, the engines behind Internet Explorer and they will just differ on the version of Windows up to Windows 8.1.

For example, on Windows 8 the engine is the same as in IE10, on Windows 8.1 is the same as in IE11. From Windows 10 on all versions (Mobile and desktop), the engine was replaced by Edge removing all the old IE stuff.

While technically you can inject your own web view on a native desktop Windows app, it's not possible to do that for mobile phones at the time of this writing.

Summary of Web View

We now know that Web View engines can be pretty different on different devices -even on the same version of the OS-. The following table shows a current state of web views with their rendering and execution engines.

Table 1-7. Web views' engines

OS	devices	versions	rendering	execution
iOS UIWebView	all	all	WebKit	JavaScriptCore
iOS WKWebView	all	8.0+	WebKit	Nitro
Android	all	2.x	WebKit	JavaScriptCore
Android	all[a]	4.0-4.3	WebKit	JavaScriptCore
Android	all	4.4+	Blink	V8
Android	Amazon Kindle Fire	4.0+	WebKit	JavaScriptCore
Android	Fire Phone	4.0+	Blink	V8
Android	some Samsung devices	4.0-4.3	WebKit	JavaScriptCore
Android	some Samsung devices	4.0-4.3	Blink	V8
Android	some HTC devices	4.0-4.3	Blink	V8
Windows	7.5+	all	Trident	Chakra
BlackBerry	all	7.0-10.2	WebKit	JavaScriptCore
BlackBerry	all	10.3+	Blink	V8
Tizen	all	all	WebKit	JavaScriptCore

[a] Excluding Amazon, Samsung and HTC devices with custom web views

I'm not mentioning Firefox OS on the last table as there are no native apps there. Because everything is a web app on that OS I don't think we should consider them having a Web View as a separate engine.

Web Browsers

When talking about the classic web, usually web browsers are the only platform the we care about. In the mobile web, it's for sure the most important category but not the only one as we saw before mentioning Web Views and Web app platforms.

Now it's the turn of browsers. I won't make a big list of mobile browser here, but just enough information to understand where users are browsing the web that might affect our performance analysis and techniques.

We have already covered the most important OSs that we will follow in this book, such as iOS, Android and Windows. And maybe you are thinking that in terms of browsers it's easy: Safari, Chrome and Internet Explorer/Edge. That's a typical mistake. That's not the whole reality.

Stats

Instead of just guessing we can check different sources that will give us some hints on the current state. Unfortunately there are several sources with different measurement methods.

If you are using an Analytics tools, such as Google Analytics, you can check information about your own website and compare it with the global data from the sources.

The sources available for checking information on mobile browsers are:

- Statcounter Global Stats: select mobile browsers
- Akamai IO DataSet: select mobile browsers
- MOVR - Mobile Overview Report by Scientia Mobile
- Adobe Digital Index Mobile Browser Share
- Market Share Reports: select device type as mobile
- Wikimedia Stats, public stats from Wikipedia and other websites

I processed all those sources to get some conclusions and by the end of 2014 the market share in terms of browsers look like the next table. The table does include WebViews when used for browsing the web, such as In-App Browsers (inside Facebook) and Pseudo-browsers (Chrome on iOS).

Data will change radically if you count or don't count tablets as mobile web share as Safari for the iPad takes a lot of traffic.

Table 1-8. Mobile Web Browsers Market Share global

Browser	OSs	engines	%
Safari and WebView	iOS	WebKit	50%
Android Browser	Android	WebKit	15%

Browser	OSs	engines	%
Chrome	Android	Blink	14%
Samsung Browser	Android	Blink	10%
Opera[a]	Mixed	Presto	5%
Internet Explorer	Windows	Trident	3%
Firefox	Android & Firefox OS	Gecko	1%
UC Browser	mixed	U3 (propietary engine)	<1%
Nokia Browser	Symbian	WebKit	<1%
Nokia Browser	Series 40	Gecko	<1%
Nokia Browser	Android (Nokia X)	Chromium	<1%
BlackBerry Browser	BBOS	WebKit	<1%
Samsung Dolfin	Samsung	WebKit	<1%
Amazon Silk	Fire OS	Blink	<1%

[a] Mostly Opera Mini

If your website or web app is targeting specific countries or regions, you should be aware that mobile browser market share might change a lot. On Asia you will see more than 20% of UC Browser, while BlackBerry Browser and Internet Explorer might count for 10% and iOS for only 15% in Latin America.

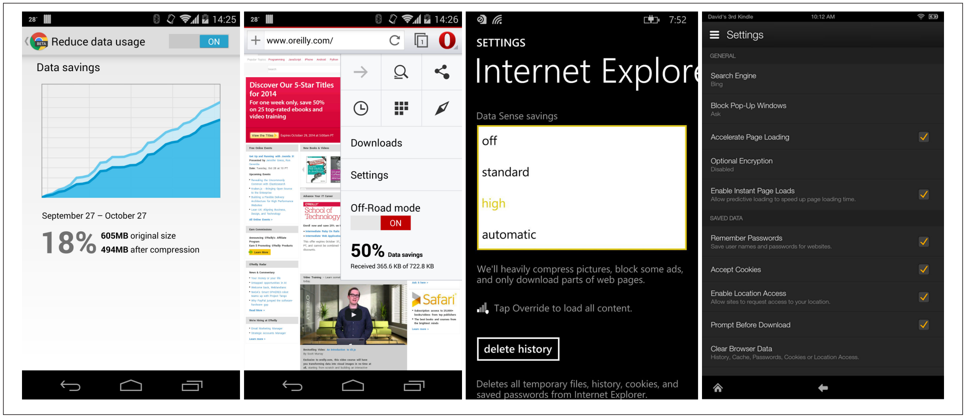
Cloud vs Direct Browsers

Direct Browsers get content directly from the website server, and cloud browsers (also known as proxied browsers), which go through a proxy server on the cloud. The proxy server usually does many of the following actions on the fly: * Reduces the content, eliminating features that are not mobile-compatible Compresses the content (images included) * Pre-renders the content, so it can be displayed in the browser faster * Converts the content, so we can see Flash video on devices with no Flash support * Encrypts the content * Caches the content for quick access to frequently visited sites

Cloud-based browsers, such as Opera Mini, consume less bandwidth, increasing performance and reducing total navigation time at the same time. From a developer's perspective, the main difference is that the device is not directly accessing our web server; it's the proxy server on the cloud that is requesting all the files from our server, rendering those files on the cloud, and delivering a compressed, proprietary result to the browser.

Google Chrome, Opera Mobile, Internet Explorer and Amazon Silk offer a cloud compression mechanism as an optional feature to their direct browser, as we can see in Figure 1-6. The feature needs to explicitly been enabled by the user (*Reduce Data Us-*

age on Chrome, *Off-Road mode* on Opera, *Compression on Silk* and *DataSense* on IE) and there is no data available on how many users are currently enabling the option.



Reduce data usage ON

Data savings

September 27 – October 27

18% 605MB original size 494MB after compression

www.oreilly.com/

Downloads

Settings

Off-Road mode ON

50% Data savings

Received 365.6 KB of 722.8 KB

SETTINGS

Internet Explore

Data Sense savings

off

standard

high

automatic

We'll heavily compress pictures, block some ads, and only download parts of web pages.

Tap Override to load all content.

delete history

Deletes all temporary files, history, cookies, and saved passwords from Internet Explorer.

David's 3rd Kindle 10:12 AM

Settings

GENERAL

Search Engine
Bing

Block Pop-Up Windows
Ask

Accelerate Page Loading ✓

Optional Encryption
Disabled

Enable Instant Page Loads ✓
Allow predictive loading to speed up page loading time.

SAVED DATA

Remember Passwords ✓
Save user names and passwords for websites.

Accept Cookies ✓

Enable Location Access ✓
Allow sites to request access to your location.

Prompt Before Download ✓

Clear Browser Data
History, Cache, Passwords, Cookies or Location Access.

Figure 1-6. Because performance is a top priority on mobile web browsing, some browsers including Chrome, Opera, IE and Silk offer you an option to enable a proxy compression option.

Pre-installed browsers

Practically every mobile device on the market today has a preinstalled browser. The average user typically doesn't install a new web browser; therefore, on each device the preinstalled browser is the most-used one. One main disadvantage of preinstalled browsers is that usually there is no way to update the browser independently from the operating system. If your device doesn't get operating system updates, usually you will not get browser updates.

iOS Browsers

On iOS there is a simple and single answer: Safari. We saw that 50% of the mobile web traffic comes from iOS and fortunately all the traffic uses the same engine.

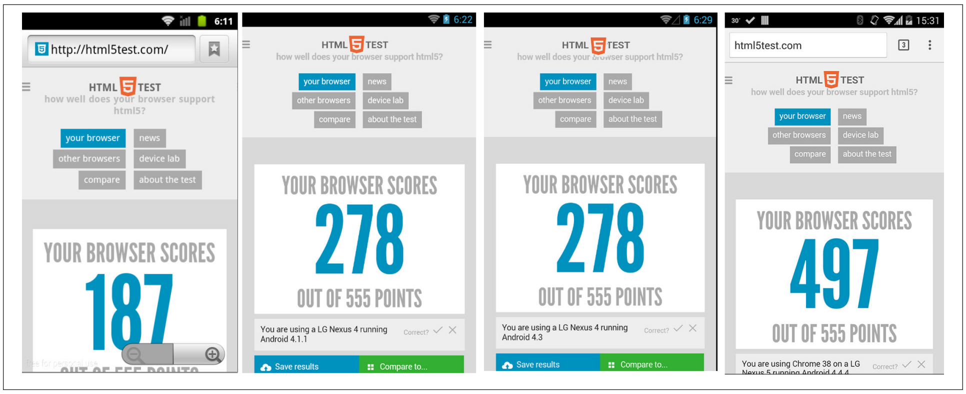
If you are not browsing the web on iOS with Safari is because you are using a pseudo-browser, such as Chrome, or an In-App Browser, such as clicking on a link inside Facebook. On any case we are using the same WebCore. The JavaScript runtime might differ as in different versions of the web view (UIWebView or WKWebView) we have JavaScriptCore or Nitro with JIT compiler.

According to MOVR - Mobile Overview Report, 5% of the iOS traffic comes from Facebook In-App Browser, 0.5% from Twitter and 6% from Chrome on iOS. Therefore, around 12% of the iOS traffic is being executed on a WebView and not inside Safari.

Android Browsers

Some developers and designers think that if we are talking about Android Browser we are talking about Google Chrome. Unfortunately, that's not the reality.

Android, as an open source operating system, doesn't include Chrome (not even today). It includes a Chromium-based web view since 4.4 and it had included a Browser up to 4.3, based purely on WebKit. This browser is usually known as *Android Browser*, *Stock Browser* or *Android WebKit*. The browser while sharing roots with Chrome on WebKit, it's a very different app in terms of performance and web compatibility as you can see in Figure 1-7.



You are using a LG Nexus 4 running Android 4.1.1

You are using a LG Nexus 4 running Android 4.3

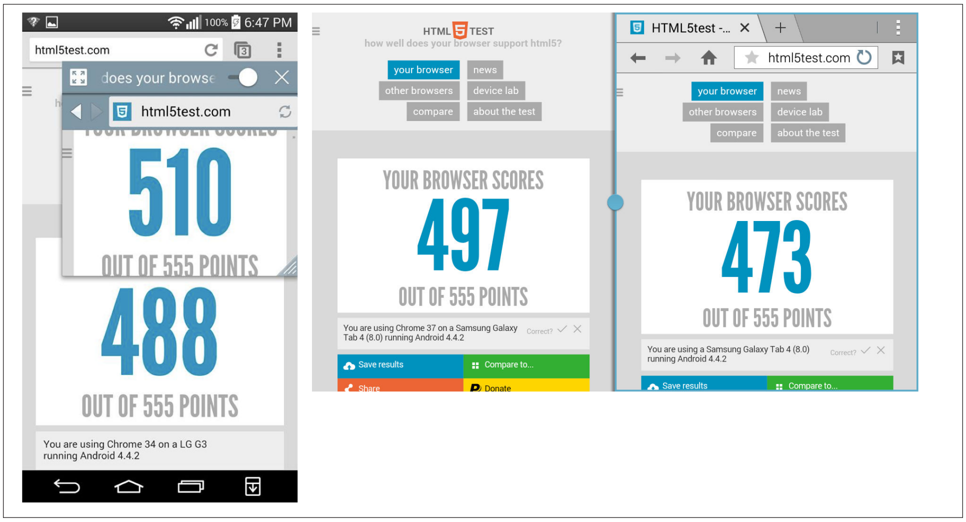
You are using Chrome 38 on a LG Nexus 5 running Android 4.4.4

Figure 1-7. Here we can see HTML5test.com results on Android Browser 2.3, 4.0, 4.3 compared with Google Chrome.

From Android 4.4, Google has decided to remove support for that browser but it didn't replace it by Chrome either (at least not technically). I know you've seen several Android 4.4+ devices with Chrome pre-installed. That is because the manufacturer (Samsung, HTC, Motorola, Sony, etc.) signed a separate agreement with Google to include Play Services and Google's apps that are not part of the Android OS, such as Maps, GMail, Inbox, Play Store and Google Chrome.

Because Google Chrome is not really part of the AOSP (Android Open Source Project) and it's part of Play Services, it can be uninstalled, re-installed and updated from the Google Play Store by the user at any time. That's not true when talking about pre-installed browsers.

Because the default Android Browser wasn't good enough, some vendors using Android as an OS including Samsung, Amazon, HTC, Nokia and Barnes and Noble had created their own custom versions of the default browser, sometimes WebKit-based (but most modern than the one used by Android), sometimes Chromium-based (using the Blink engine).



You are using Chrome 34 on a LG G3 running Android 4.4.2

You are using Chrome 37 on a Samsung Galaxy Tab 4 (8.0) running Android 4.4.2

You are using a Samsung Galaxy Tab 4 (8.0) running Android 4.4.2

Figure 1-8. Some manufacturers, as in this case LG and Samsung, include a pre-installed browser that is not Android Browser not Chrome. They are today usually based on Chromium but as you can see here with different compatibility with the original Chrome.

It's not all, because some manufactures and some carriers around the globe made agreements with Opera or Firefox to replace the default browser with those versions.

Some devices, such as the Samsung Galaxy series include two different browsers pre-installed, such as Chrome thanks to a Google Play agreement and the Samsung Browser (*Internet* as the name of the icon) (also Chromium-based on modern devices). The user has the choice to use one or the other.

The following table gives as an idea of how complicated the pre-installed browsers look like on Android. Remember, most users will use just that browser.

Table 1-9. Pre-installed browsers on Android

Browser	devices	version	engine	updatable
Android Browser	all[a]	1.0-4.3	WebKit	no
Google Chrome	manufacturers with a Play Services agreement	4.2+	Blink	yes
Samsung Browser	Galaxy S3 and others	4.2+	WebKit	no
Samsung Browser	Galaxy S4 and others	4.2+	Blink	no
Nokia Browser	Nokia X, X+	4.1, 4.3	Blink	yes
HTC Browser	some	4.2+	Blink	no
LG Browser	some	4.2+	Blink	no
Silk	Kindle Fire	4.2+	WebKit or Blink	no
Silk	FirePhone	4.2+	Blink	no
Opera	based on carrier	4.0+	Blink	yes
Firefox	Kobo, Gigabyte devices	4.0+	Gecko	yes

[a] excepting those whose manufacturer replaced by a custom browser

Samsung and web runtimes

Because the Android WebView and Browser were old and slow before Android 4.4 Samsung has decided to change them. However, there is no single or simple rule here.

To give you some examples, on the Galaxy SIII the WebView and the Browser are WebKit-based but not the ones provided by AOSP -Android Open Source Project-. On the Galaxy SIV, they still have the same WebKit-based WebView, but the browser was upgraded to a Chromium engine (version 18, upgraded to 28). Other devices that were shipped after the Galaxy SIV were still on WebKit.

Even when Samsung is using a version of Chromium, that doesn't mean that features and bugs are the same as in Chrome on that same version, because Samsung is enabling and disabling different experimental features.

All the new Android phones, phablets and tablets shipped from 2014 have any kind of Chromium-based Samsung Browser, but there is no public information from the company about that.

Some people is treating Android Browser as the modern Internet Explorer 6, as a slow, non-compatible browser that is difficult to get rid off the market and that we need to still support. In July, 2013 I wrote on Article with the title Android and the eternal dying mobile browser.

Android users have several options from the Google Play Store in terms of downloading browsers, including Firefox (Gecko), Opera Mobile (Blink), Opera Mini (cloud-, Presto-based), UC Browser (cloud-based) and Dolphin (Blink). Unfortunately, stats show that only few users use additional browsers.

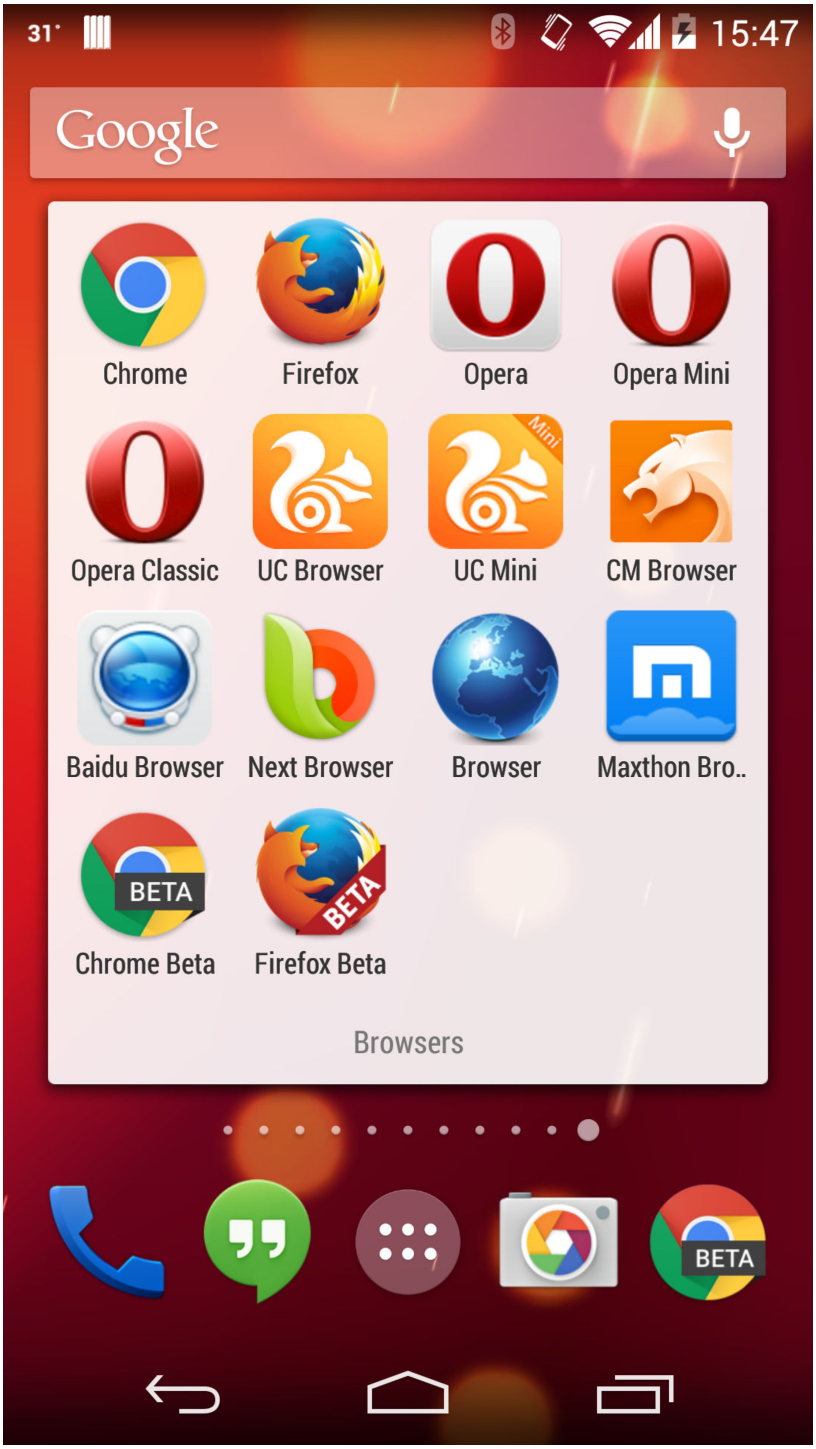


Figure 1-9. On Android, the user has the option to install several browsers on the same

device from the Play Store.

Windows Browsers

On Windows mobile devices, Internet Explorer on different versions (based on the OS version) was the only browser available by default up to Windows 8.1 and it was replaced by Microsoft Edge since Windows 10.

Some cloud-based browser are available to download from the store, including Xpress Browser and UC Browser.

The differences

In this chapter, we've covered the main differences that the mobile web has compared to classic web engines. It's important to not underestimate the mobile web and to test, debug and measure on most scenarios as we probably are not an average user in terms of mobile device, cellular network and browser.

CHAPTER 2

Where to measure performance

Measuring speed is critical if we want to detect our problems and improve performance. The first question that appears when talking about measurement is the target. Where should we measure?

Your first thought may be to measure on desktop. There are several tools available for classic web performance optimization. However, as we saw in last chapter, mobile browsers, hardware and networks are different. Therefore, measuring on a classic browser will lead to false ideas of performance.

There are several measurements available in web performance that we will cover during the rest of the book. In this section we will discuss the options available that we will mention later.

When measuring web performance on mobile devices, sometimes the absolute values are not the most important data, but the comparison with different techniques and the relative difference when you are applying web performance optimization techniques because environments are really dynamic on the market, so saying that your website can be loaded in 2 seconds in a cellular phone means nothing, because it can be much slower on 2G networks on unreliable 3G networks or on some different mobile browsers.

There are several companies offering solutions to monitor performance of websites, web apps and hybrid apps' performance on real environments. In this chapter we'll focus on solutions that we can use from our side.

Simulators & Emulators

Generally speaking, an *emulator* is a piece of software that translates compiled code from an original architecture to the platform where it is running. It allows us to run an operating system and its native applications on another operating system. In the mobile development world, an emulator is a desktop application that emulates mobile device hardware and a mobile operating system, allowing us to test and debug our applications and see how they are working. The browser, and even the operating system, is not aware that it is running on an emulator, so we can execute the same code that we would execute on the real device.

On next chapter we'll analyze some HTML5 APIs that will help us measure and log performance on real user's devices and connections. Those cases will help us getting data from the wild real world out there.

In terms of performance measurement, because there is a translation between the original code and our host machine, the measurement won't be accurate. However, we can definitely use them for comparisons and to get accurate waterfall charts of how the browser is rendering our page.

Emulators are created by manufacturers and offered to developers for free, either standalone or bundled with the Software Development Kit (SDK) for native development.

There are also operating system emulators that don't represent any real device hardware (for example, a Galaxy S5), but rather the operating system as a whole. These exist for Windows Phone and Android.

The definitions of *emulator* and *simulator* in this book are defined by majority. However, some vendors are mixing those names. For example, BlackBerry Simulators are real virtual machines, so they are under our emulator concept.

On the other hand, a *simulator* is a less complex application that simulates some of the behavior of a device, but does not emulate hardware and does not work over the real operating system. These tools are simpler and less useful than emulators. A simulator may be created by the device manufacturer or by some other company offering a simulation environment for developers.

In mobile browsing, there are simulators that create a skin over a typical desktop browser (such as Firefox, Chrome, or Safari) without these browsers' rendering engines.

For performance purposes, simulators -as tools that looks like X but they are not really the code on X- are not interesting. They are usually running on a classic desktop browser without using the same behavior of their real counterpart

Android

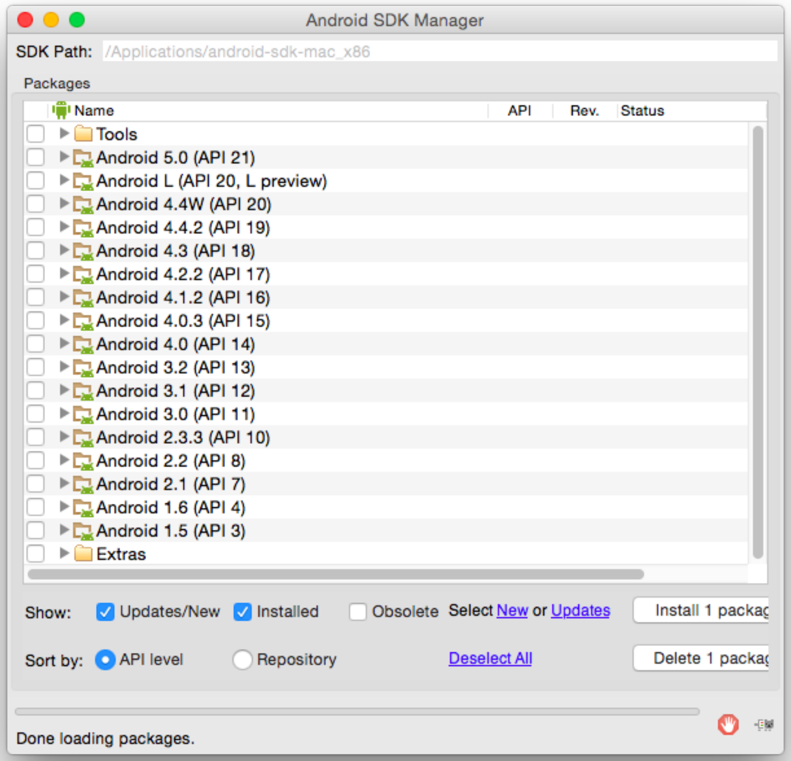
As we covered in last chapter, there are several browsers and web views available on the Android OS and that means that there are different techniques to measure performance on emulators and simulators.

Android Browser

The Android default browser available until version 4.3 can be emulated through the official Android OS emulator. Because this emulator is based on AOSP -Android Open Source Project- it included the standard Android Browser until version 4.3. From 4.4, the AOSP project still includes the Browser app inside but using the Web View -that is Chromium-based- and it shouldn't be used for performance, because it's a different engine.

The Android emulator is available in conjunction with the SDK to create native applications for Android. You can download it for free from the Android Developer page; the base SDK and the different Android OS versions are available separately.

The Android emulator is available for Windows, Mac OS X, and Linux. Once you've downloaded it, create a folder for the contents on your hard drive and unzip the package. On Windows, there is an installer version that will do the work for you.



SDK Path: /Applications/android-sdk-mac_x86

Name	API	Rev.	Status
Tools			
Android 5.0 (API 21)			
Android L (API 20, L preview)			
Android 4.4W (API 20)			
Android 4.4.2 (API 19)			
Android 4.3 (API 18)			
Android 4.2.2 (API 17)			
Android 4.1.2 (API 16)			
Android 4.0.3 (API 15)			
Android 4.0 (API 14)			
Android 3.2 (API 13)			
Android 3.1 (API 12)			
Android 3.0 (API 11)			
Android 2.3.3 (API 10)			
Android 2.2 (API 8)			
Android 2.1 (API 7)			
Android 1.6 (API 4)			
Android 1.5 (API 3)			
Extras			

Show: [✓ Updates/New] [✓ Installed] Obsolete

Sort by: [✓ API level] Repository

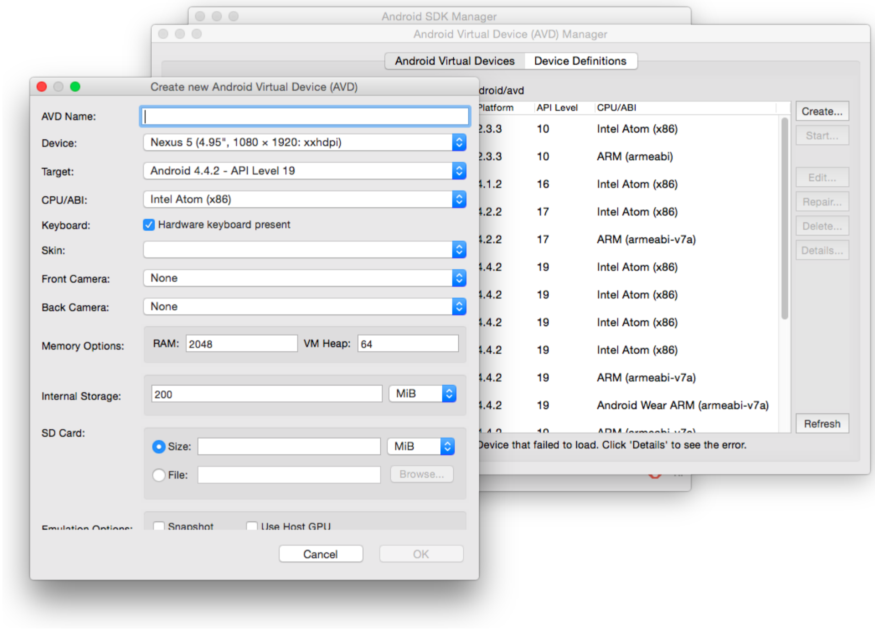
Done loading packages.

Figure 2-1. The Android SDK manager allows us to download different versions of Android OS to emulate

In the folder where you extracted the package, there is an **`android`** terminal command on Mac OS X/Linux and an **`SDK Setup.exe`** application for Windows that opens the Android SDK Manager shown in Figure 2-1, where you can download and configure Android platforms (known as packages or targets) after installing the base SDK.

Android Emulator is really slow, which might affect our perception of performance. We can increase performance if we have an Intel compatible CPU in our host computer using a free tool known as Intel Harware Acceleration Execution Manager (HAXM), or switching to Genymotion.

You can download as many packages as you want, one per operating system version; you can even download vendor-specific emulators, such as for the Galaxy devices or Amazon Fire Phone. Try to download different releases of every Android version up to 4.3 which was the last version with Android Browser, such as Android 2.3.3, Android 4.1, and Android 4.3.



Create new Android Virtual Device (AVD)

AVD Name:	
Device:	Nexus 5 (4.95", 1080 × 1920: xxhdpi)
Target:	Android 4.4.2 - API Level 19
CPU/ABI:	Intel Atom (x86)
Keyboard:	✓ Hardware keyboard present
Skin:	
Front Camera:	None
Back Camera:	None
Memory Options:	RAM: 2048 VM Heap: 64
Internal Storage:	200 MiB
SD Card:	● Size: MiB ○ File:
Emulation Options:	☐ Snapshot ☐ Use Host GPU

droid/avd

Platform	API Level	CPU/ABI
2.3.3	10	Intel Atom (x86)
2.3.3	10	ARM (armeabi)
4.1.2	16	Intel Atom (x86)
4.2.2	17	Intel Atom (x86)
4.2.2	17	ARM (armeabi-v7a)
4.4.2	19	Intel Atom (x86)
4.4.2	19	Intel Atom (x86)
4.4.2	19	Intel Atom (x86)
4.4.2	19	Intel Atom (x86)
4.4.2	19	ARM (armeabi-v7a)
4.4.2	19	Android Wear ARM (armeabi-v7a)
4.4.2	19	ARM (armeabi-v7a)

Device that failed to load. Click 'Details' to see the error.

Figure 2-2. Once you have the image of the OS version you want to emulate you need to create an Android Virtual Device (AVD), an instance of an emulator

Other solution for emulating Android Browser is to install Genymotion. Genymotion is a an alternate Android emulator based on VirtualBox and AOSP. It's roughly 10 times faster than the original Android SDK and it includes the same version of Android Browser.

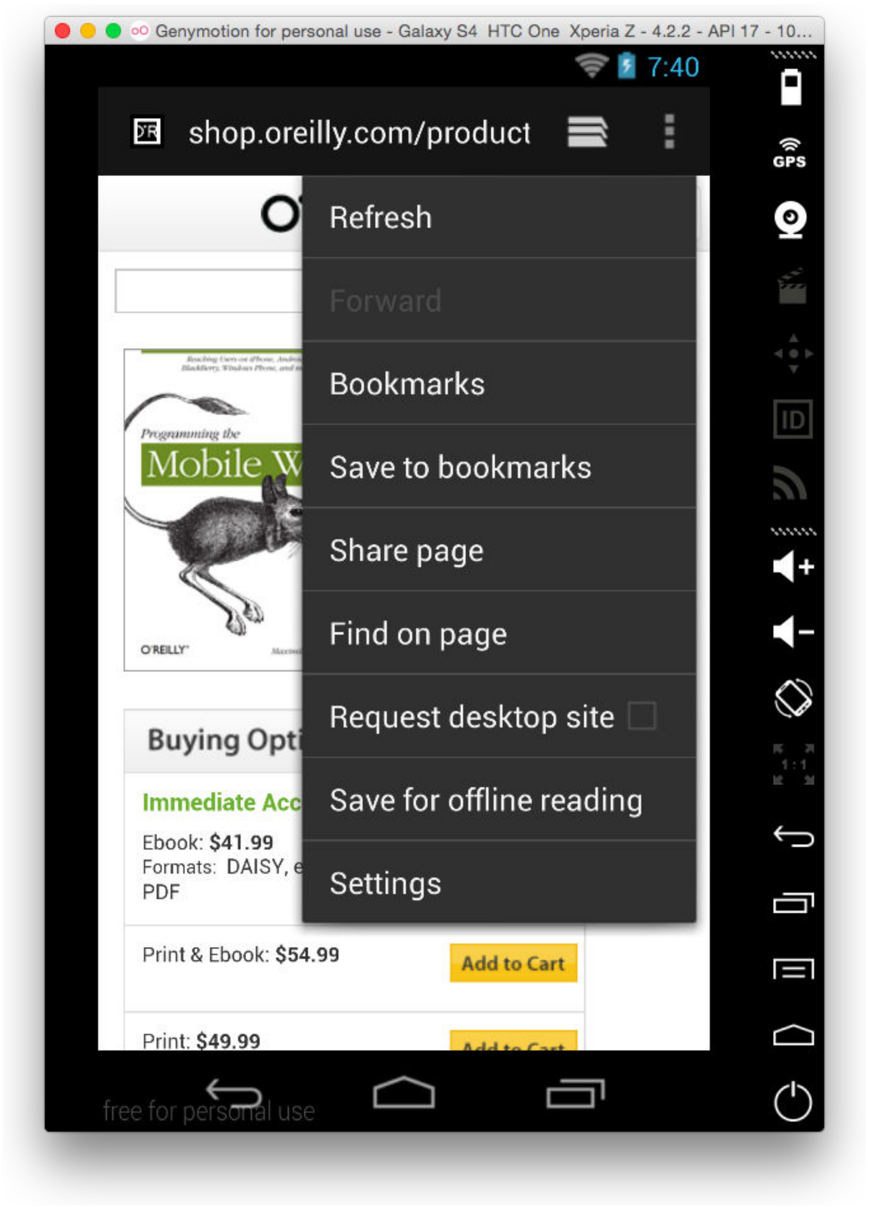
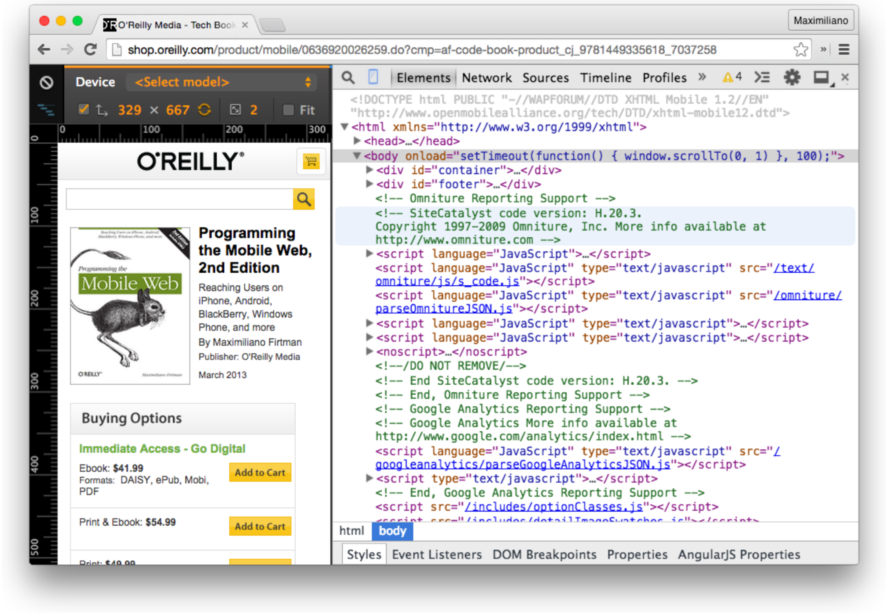


Figure 2-3. Genymotion is a faster Android emulator; it can be used to emulate Android Browser. You can also install Firefox or Opera, but not Google Chrome

Google Chrome

Unfortunately Google Chrome for Android requires Play Services, a set of tools that are outside of AOSP. That means that we won't find Google Chrome on them and we can't install it.

We can emulate Chrome for Android using the Device Mode & Mobile Emulation tools available on Chrome on desktop, on Mac, Linux, Windows and Chrome OS. If you open developer tools, you will see a small device icon at the top left corner as you can see in Figure 2-4



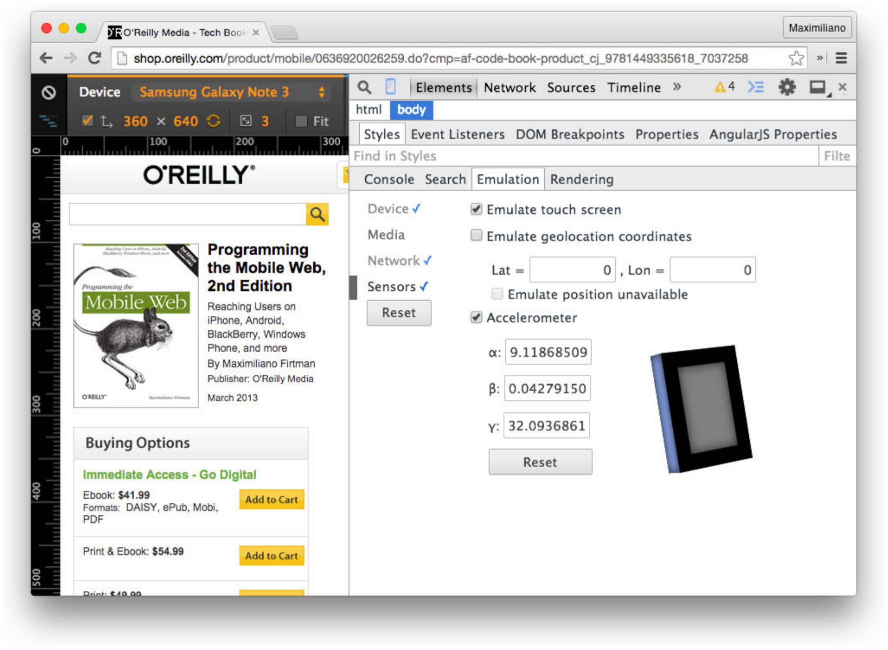
O'Reilly Media - Tech Book
Maximiliano
shop.oreilly.com/product/mobile/0636920026259.do?cmp=af-code-book-product_cj_9781449335618_7037258
Device
<Select model>
329 × 667
2
Fit
Elements
Network
Sources
Timeline
Profiles
O'REILLY
Programming the Mobile Web, 2nd Edition
Reaching Users on iPhone, Android, BlackBerry, Windows Phone, and more
By Maximiliano Firtman
Publisher: O'Reilly Media
March 2013
Buying Options
Immediate Access - Go Digital
Ebook: $41.99
Formats: DAISY, ePub, Mobi, PDF
Add to Cart
Print & Ebook: $54.99
Add to Cart
html
body
Styles
Event Listeners
DOM Breakpoints
Properties
AngularJS Properties

```
<!DOCTYPE html PUBLIC "-//WAPFORUM//DTD XHTML Mobile 1.2//EN"
"http://www.openmobilealliance.org/tech/DTD/xhtml-mobile12.dtd">
▼<html xmlns="http://www.w3.org/1999/xhtml">
  ▶<head>…</head>
  ▼<body onload="setTimeout(function() { window.scrollTo(0, 1) }, 100);">
    ▶<div id="container">…</div>
    ▶<div id="footer">…</div>
     <!-- Omniture Reporting Support -->
     <!-- SiteCatalyst code version: H.20.3.
     Copyright 1997-2009 Omniture, Inc. More info available at
     http://www.omniture.com -->
    ▶<script language="JavaScript">…</script>
     <script language="JavaScript" type="text/javascript" src="/text/
     omniture/js/s_code.js"></script>
     <script language="JavaScript" type="text/javascript" src="/omniture/
     parseOmnitureJSON.js"></script>
    ▶<script language="JavaScript" type="text/javascript">…</script>
    ▶<script language="JavaScript" type="text/javascript">…</script>
    ▶<noscript>…</noscript>
     <!--/DO NOT REMOVE/-->
     <!-- End SiteCatalyst code version: H.20.3. -->
     <!-- End, Omniture Reporting Support -->
     <!-- Google Analytics Reporting Support -->
     <!-- Google Analytics More info available at
     http://www.google.com/analytics/index.html -->
     <script language="JavaScript" type="text/javascript" src="/
     googleanalytics/parseGoogleAnalyticsJSON.js"></script>
    ▶<script type="text/javascript">…</script>
     <!-- End, Google Analytics Reporting Support -->
     <script src="/includes/optionClasses.js"></script>
```

Figure 2-4. With Google Chrome you can emulate Chrome on Android devices and you can simulate other mobile phones or tablets.

Google Chrome emulation can be considered an emulator for Chrome on Android and a simulator when you are choosing other non-Android device, as on those cases we are going to still work on Chrome while thinking we are on a different browser (such as Safari on iOS or IE on Windows).

To enable device mode, press the Toggle Device Mode button (it looks like a mobile phone). When device mode is enabled this icon will turn blue. You can simulate different mobile scenarios, but rememeber the engine will be the same as the one running at Chrome on Android.


shop.oreilly.com/product/mobile/0636920026259.do?cmp=af-code-book-product_cj_9781449335618_7037258

Device: Samsung Galaxy Note 3 — 360 × 640, 3

- [✓] Emulate touch screen
- [ ] Emulate geolocation coordinates
  - Lat = 0, Lon = 0
  - [ ] Emulate position unavailable
- [✓] Accelerometer
  - α: 9.11868509
  - β: 0.04279150
  - γ: 32.0936861

Figure 2-5. Once enabled you can set different properties to match the simulation you want while having access to the full Chrome developer tools.

When working with the Device Mode and Mobile Emulation of Chrome, a good idea is to dock the developer tools window to the right.

Samsung Browser

The Chromium-based Samsung Browser available on most of their Android devices, such as in the Galaxy series, can't be emulated. Samsung doesn't provide the code, even if you download the SDK add-ons from Samsung.

Because, it's Chromium-based we can assume that most stuff will work similar to Chrome on the same version but we are never sure.

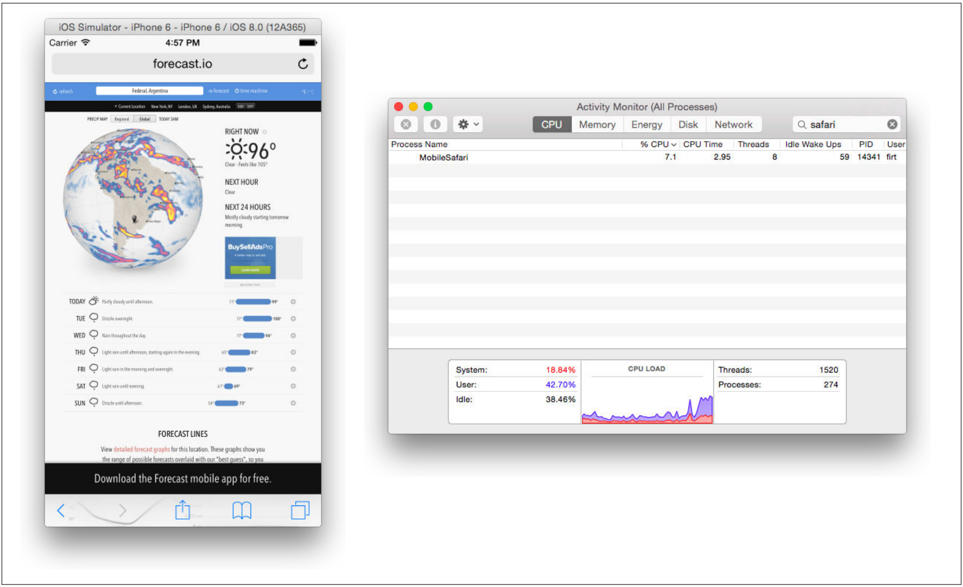
The best approach for this browser is then using a real device.

Amazon Silk, LG and other custom browsers

Following Samsung, Amazon doesn't provide any way to test the Silk browser -the one available on Kindle Fire and Fire Phone devices. The same happens with LG and the browser available on some of its devices, such as the G3.

Safari on iOS

Available for only Mac OS X, the iOS Simulator (shown in Figure Figure 2-6) offers a free simulation environment for the iPhone and iPad, including the mobile browser Safari. It is not an emulator, so it does not really provide a hardware emulation experience and is not a true performance indicator. However, the Safari code that is being executed in this simulated environment is the real MobileSafari.app so we can think on it as a Safari emulator running over an iOS simulator.


iOS Simulator - iPhone 6 - iPhone 6 / iOS 8.0 (12A365)
Carrier
4:57 PM
forecast.io
RIGHT NOW
96°
NEXT HOUR
NEXT 24 HOURS
FORECAST LINES
Download the Forecast mobile app for free.
Activity Monitor (All Processes)
CPU
Memory
Energy
Disk
Network
safari
CPU LOAD

Process Name	% CPU	CPU Time	Threads	Idle Wake Ups	PID	User
MobileSafari	7.1	2.95	8	59	14341	firt

System:	18.84%
User:	42.70%
Idle:	38.46%
Threads:	1520
Processes:	274

Figure 2-6. The iOS simulator runs the real Safari and Web app code on your Mac with an iOS skin. While it runs as a simulation, the real Safari on iOS code and WebCore is running your code.

The iOS Simulator is included with the SDK for native development, available for free at the Mac App Store (search for Xcode) or at from Apple's website. The SDK may take a while to download, because it's about 1.5 GB. You will always download the latest version of the operating system and can then add previous versions (such as 8.x), in which case you can switch between versions using the Hardware→Version menu option (Figure 2-7).

The iOS Simulator can also be used to measure performance for Web apps and Web View-based solutions.

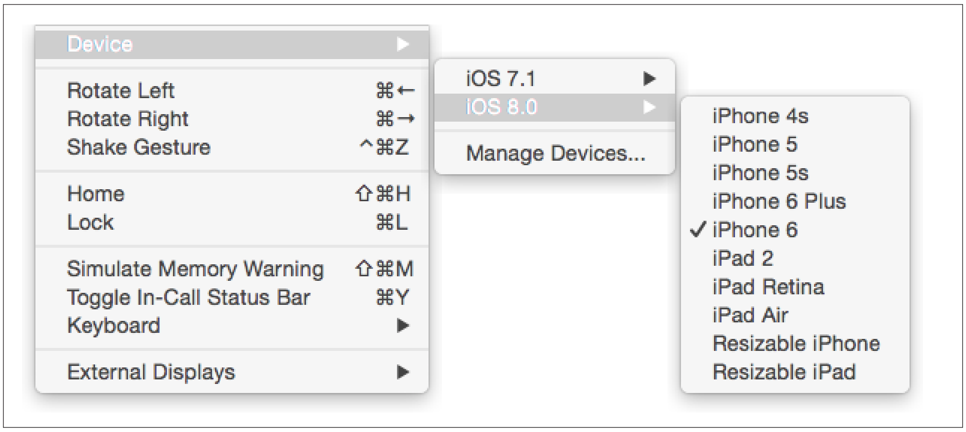
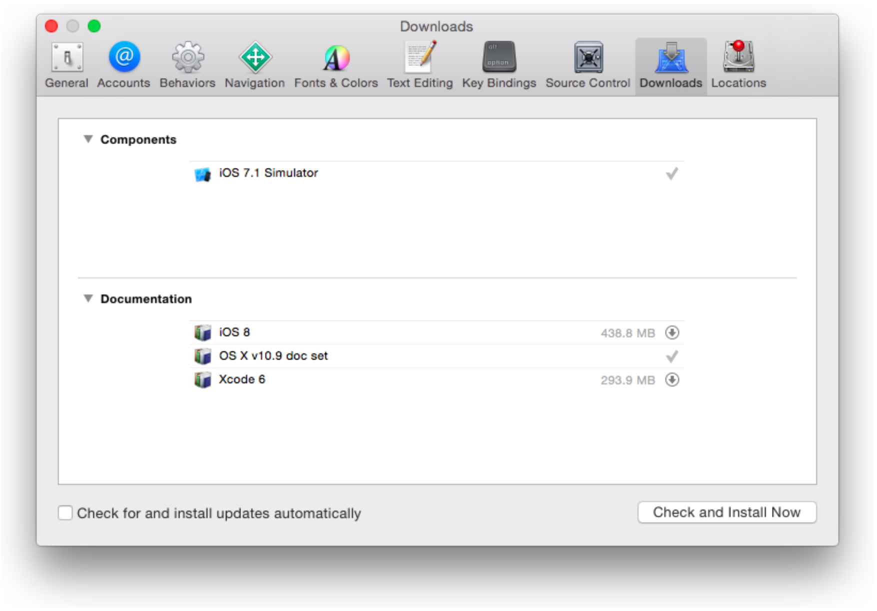


Figure 2-7. In the Hardware menu you can select which iPad or iPhone do you want to simulate.

To download a previous version of the operating system to the simulator, you need to open the Xcode app, open Preferences, and select Downloads, as seen in Figure Figure 2-8.



Components

iOS 7.1 Simulator	✓

Documentation

iOS 8	438.8 MB
OS X v10.9 doc set	✓
Xcode 6	293.9 MB

Figure 2-8. You can download previous versions of iOS simulator from the Preferences menu.

You can only use the iOS Simulator with Safari on a Mac OS X desktop computer. If you are in other OS, you can use the cloud-based service BrowserStack. It includes emulation of Safari on iOS, the Android browser, and Opera through different device profiles without the need of any local installation.

Internet Explorer

If you want to test your applications on Windows Phone, you can download the free Windows Phone SDK or buy a license of Visual Studio. The Windows Phone emulator (Figure Figure 2-9) comes with the SDK and includes the current version of Internet Explorer to test web content.

Figure 2-9. The Windows Phone emulator includes Internet Explorer mobile runtime

and it can be used for testing.

The Windows Phone emulator from version 8 is compatible with only Windows Windows 8 Pro and requires a graphic driver with WDDM 1.1 support. You can check your hardware specifications to verify whether your graphic driver is compatible. If not, you will see the emulator, but you will see only a white page when trying to load Internet Explorer.

If you want to emulate Windows 8 for tablets, you have two options: Use your own Windows 8 for desktop environment.

Use the Windows Simulator included with Visual Studio for Windows 8 (even with Express, a free version of the IDE). The Windows Simulator works only on Windows 8 desktop machines; it emulates a tablet touch environment, where you can emulate touch gestures, geolocation, and different screen sizes and orientations Figure 2-10.

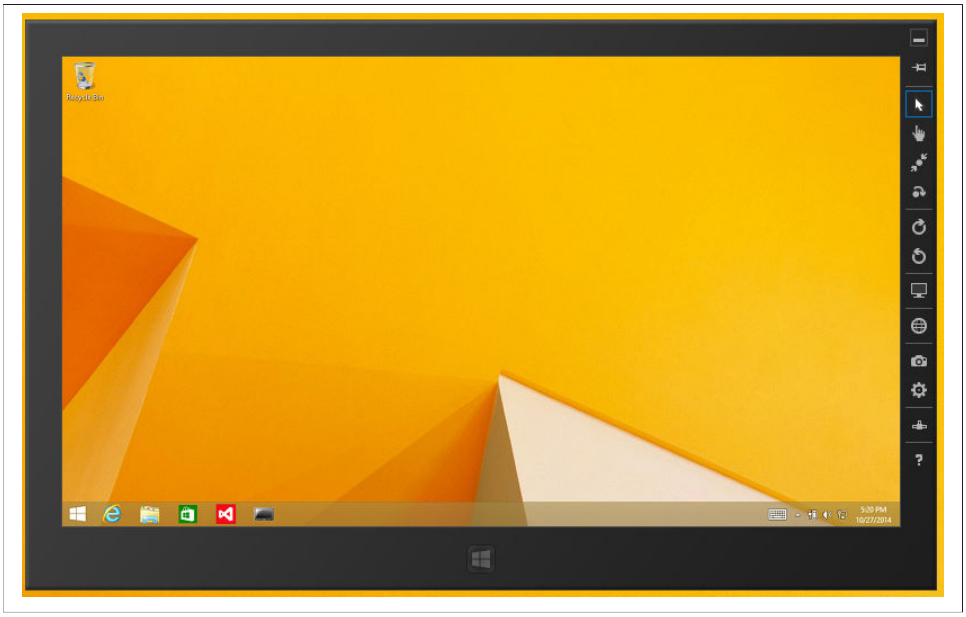


Figure 2-10. The Windows emulator allows us to test on Windows tablets, but it's not emulating th real hardware as in the Phone version.

Microsoft Edge

The new browser from Microsoft is available only on Windows 10. At the time of this writing Windows 10 is available only as a technology preview. To test your websites on it you need to download a Windows 10 pre-release version available at XXX.

Other Browsers

BlackBerry, Firefox OS, Tizen and other OSs have emulators or simulators available. Check *http://emulato.rs* for more links.

Real devices

There is nothing like real devices when measuring mobile websites. You will find differences not only in performance but also in behavior, like when you use your fingers to navigate and not a precise mouse pointer.

And while creating your own testing lab is ideal, it's also expensive and needs to be updated frequently. At the time of this writing, I currently have around 45 devices for testing.

If you have a limited budget, you should try to buy one key device per platform, and if you are targeting tablets you should get one. When you are doing the selection, don't buy just expensives phones but a mix. For example, you can buy:

- And old Android phone (it can be 2.3 or 4.0)
- A very cheap new Android phone
- An average Android phone
- A high-end Android phone
- An old iPhone (such as a 4S)
- A new iPhone
- A Windows Phone device

One of those Android phones should be a Galaxy S3 or newer as they have the Chromium-based custom browser.

Also try to get different SIM cards from different providers so you can test on real scenarios.

Figure 2-11. Having several devices is always a good idea when you are doing performance optimization.

Open Device Lab

Because we can't have every device and keep our lab updated, a great community-based project has appeared in different cities in the world: Open Device Lab. The idea is simple: to offer a physical place around the globe where you can go and test on real devices for free.

At the time of this writing, more than 130 open device labs are opened across 32 countries.

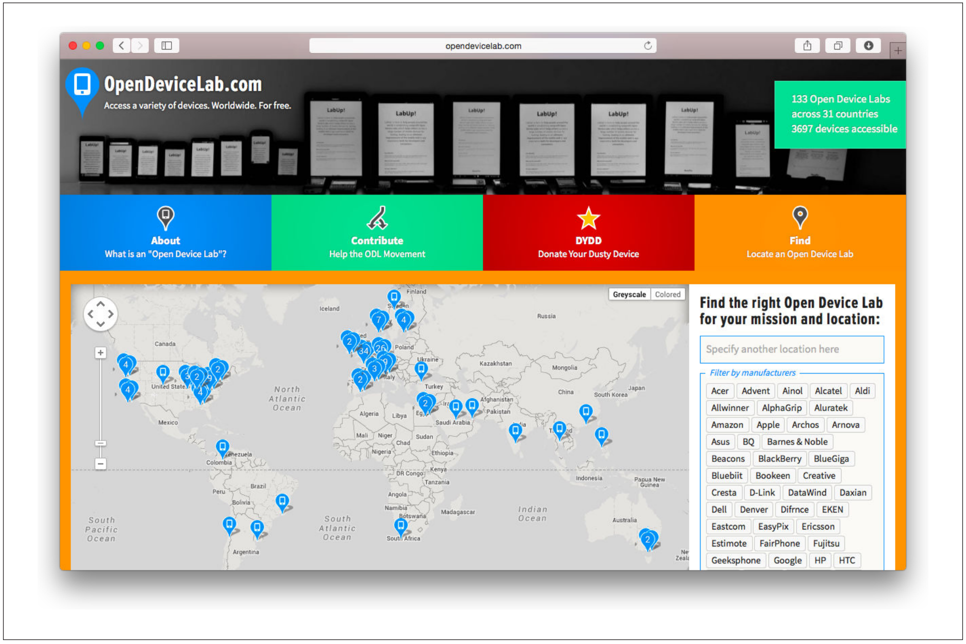


Figure 2-12. When you can't have every device at home/office, Open Device Lab will help you as a community-based solution to share testing devices.

Cloud-based services

If you can't afford buying new devices and you don't have any Open Device Lab around your city, we still have one option: remote labs on the cloud.

A *remote lab* is a web service that allows us to use a real device remotely without being physically in the same place. It is a simple but very powerful solution that gives us access to thousands of real devices, connected to real networks all over the world, with a single click. You can think of it as a remote desktop for mobile phones.

There are three kinds of remote lab solutions for mobile devices:

- Software-based solutions, using a resident application on the device that captures the screen, sends it to the server, and emulates keyboard input or touches on the screen
- Hardware-based solutions, using some technology (magic, I believe) to connect the server to the hardware components of the device (screen, touch screen, keypad, lights, audio, and so on)
- Mixed solutions, having some hardware connection, some software additions, and maybe a video camera for screen recording

For performance measurement we need to remember they are real phones over real networks so they are as accurate as we need.

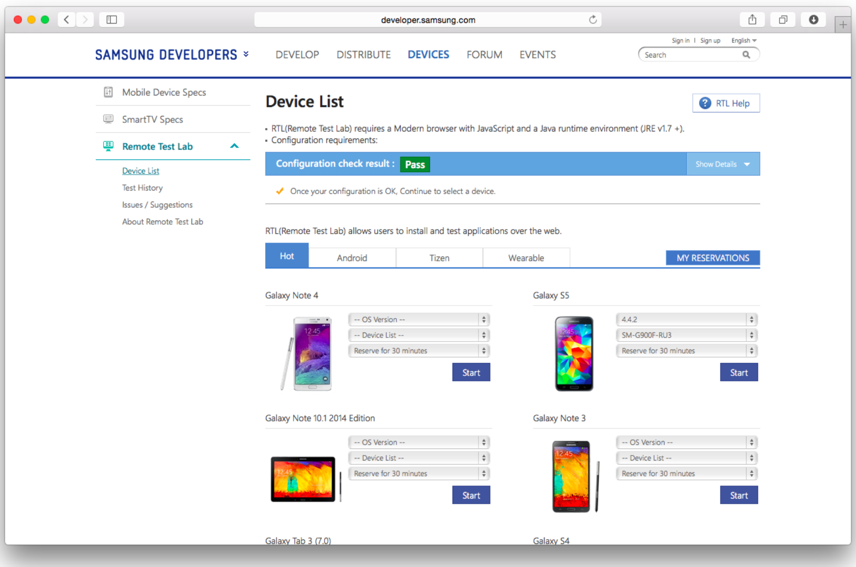
Some remote labs are connecting their phones to Wi-Fi automatically. Therefore we can't test and measure using the cellular network with the real latency. Verify if you are able to browser through cellular on those services.

Samsung Remote Test Lab

Samsung offers a free remote lab web service called Remote Test Lab (RTL). RTL, shown in Figure 4-19, includes Android smartphones, Android tablets, smartwatches and Tizen phones at the time of this writing. The devices don't have SIM cards, though, so you can test only WiFi connections (not cellular).

It's a great opportunity to test and measure on the Samsung Browser as this browser can't be emulated.

We can also upload APKs (Android native apps) that are using the Web View to test on the custom Web Views on these devices.



Device List

- RTL(Remote Test Lab) requires a Modern browser with JavaScript and a Java runtime environment (JRE v1.7 +).
- Configuration requirements:

Configuration check result : Pass

✓ Once your configuration is OK, Continue to select a device.

RTL(Remote Test Lab) allows users to install and test applications over the web.

Galaxy Note 4: -- OS Version --; -- Device List --; Reserve for 30 minutes

Galaxy S5: 4.4.2; SM-G900F-RU3; Reserve for 30 minutes

Galaxy Note 10.1 2014 Edition: -- OS Version --; -- Device List --; Reserve for 30 minutes

Galaxy Note 3: -- OS Version --; -- Device List --; Reserve for 30 minutes

Galaxy Tab 3 (7.0)

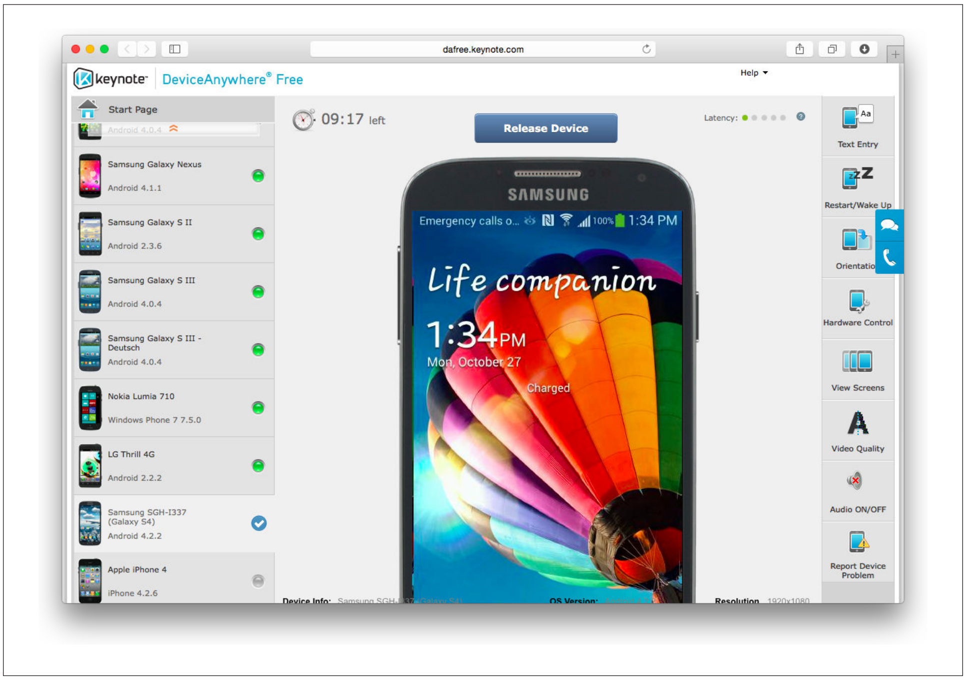
Galaxy S4

Figure 2-13. Samsung Remote Test Lab allows us to use for free, several Android- and Tizen-based devices through the network.

Until mid-2014, Nokia had a cloud-based testing solution for Windows Phone and other devices. This service was cancelled by Microsoft a few months after the acquisition and there was no replacement announced.

Keynote Mobile Testing

Formerly known as DeviceAnywhere, Keynote Mobile Testing is one of the leaders in Virtual Labs.



dafree.keynote.com
keynote DeviceAnywhere® Free
Help
Start Page
09:17 left
Release Device
Latency:
Samsung Galaxy Nexus Android 4.1.1
Samsung Galaxy S II Android 2.3.6
Samsung Galaxy S III Android 4.0.4
Samsung Galaxy S III - Deutsch Android 4.0.4
Nokia Lumia 710 Windows Phone 7 7.5.0
LG Thrill 4G Android 2.2.2
Samsung SGH-I337 (Galaxy S4) Android 4.2.2
Apple iPhone 4 iPhone 4.2.6
Text Entry
Restart/Wake Up
Orientation
Hardware Control
View Screens
Video Quality
Audio ON/OFF
Report Device Problem
SAMSUNG
Life companion
1:34 PM
Mon, October 27
Charged

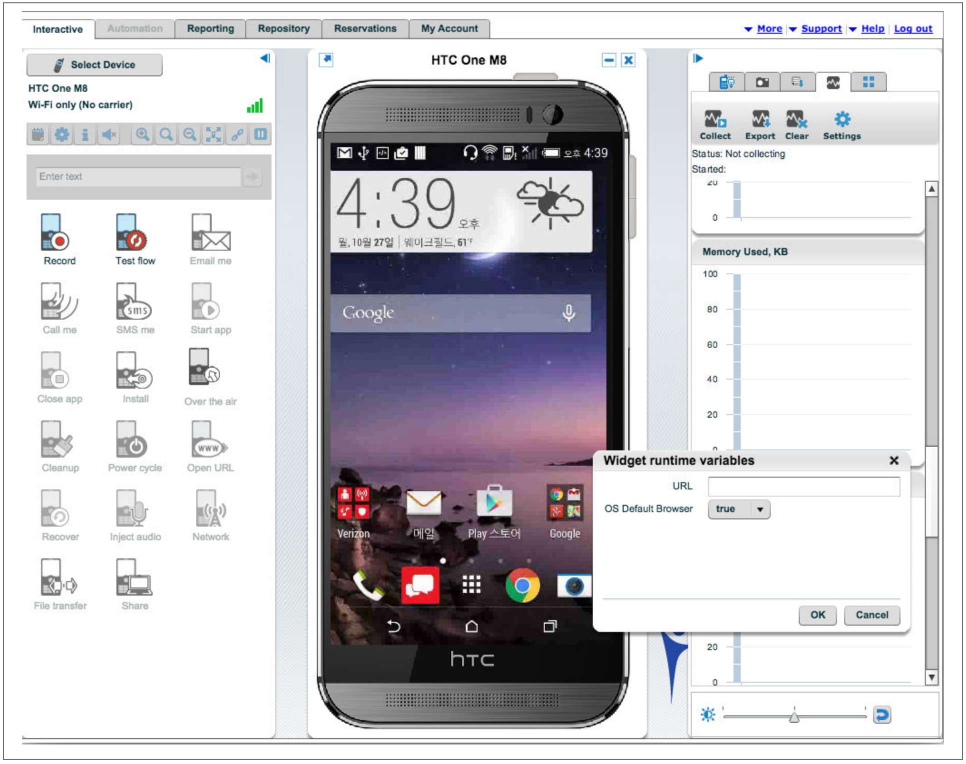
Figure 2-14. With Keynote Mobile Testing we have the option to use hundreds of different devices on different real networks over the world.

The company offers a product called Test Center Developer, with different price models depending on the package. DeviceAnywhere Test Center offers more than 2,000 devices (iOS, Android, Nokia, Motorola, Sony Ericsson, Samsung, BlackBerry, LG, Sanyo, Sharp, HTC, and more) connected to more than 30 live networks all over the world.

There are different commercial plans available and a free version specially targeting web developers. On the free version you can use a limited amount of devices (iOS, Android, Windows and BlackBerry available) for 10-minutes sessions only. It's suitable for quick testings on websites, web apps and hybrid apps.

Perfecto Mobile

Perfecto Mobile is a company offering a software/hardware hybrid solution for mobile testing, shown in Figure Figure 2-15. Perfecto Mobile uses a video camera for screen recording on some devices. A good point for Perfecto Mobile is that the whole environment is built on top of the Adobe Flash Player, so you don't need to install anything, and it works from any desktop browser. You can try the system by registering for a free trial; it will be activated in minutes.



Widget runtime variables

URL

OS Default Browser: true

Figure 2-15. Perfecto Mobile uses an in-browser solution to allow us to use several devices on their cloud.

With this service, you have access to the whole list of devices and carriers from the same pricing policy. The devices are on real networks in Canada, Israel, the US, the UK, and France.

AppThwack

AppThwack is a commercial service targeting native apps and web apps.

The service will test your app automatically on 300+ unique, non-rooted phones and tablets and it will give you a performance report. If you want to test a web app, you will provide a URL; if you are testing a hybrid app you will upload an APK.

When you sign up for a free acount, you will get 100 minutes of usage for free.

Connections

When thinking about where are we going to measure performance, we shouldn't forget about the network differences. Having real cellular SIMs on different networks always helps to test everything in different conditions.

We should always measure over 4G, 3G, 2G and WiFi networks under different conditions to get a more accurate idea of how the experience looks like from a user's perspective.

Some performance tools also allows you to measure using a cellular connection of your desktop. We can do this with a laptop with 3G/4G, buying a hardware cellular modem for our computer, or using a tethered connection from our phone. The latter is not always accurate as some carriers might change speed when the connection is tethered and we are also adding more latency from the phone to our computer through the Ad-Hoc Wi-Fi connection.

Connection Simulators

Besides using real cellular connections, we can also use some tools to throttle our connection to simulate other kind of networks, including bandwidth, latency and packet loss.

With connection simulators we can slow down our Internet connection and simulate a real 2G, 3G, or 4G connection to get a better idea of performance and how our website is reacting.

Desktop simulators

These tools run on our desktop computer and they user our current Wi-Fi or LAN connection throttling -slowing down- it so we can simulate a mobile connection's bandwidth, latency and packet loss.

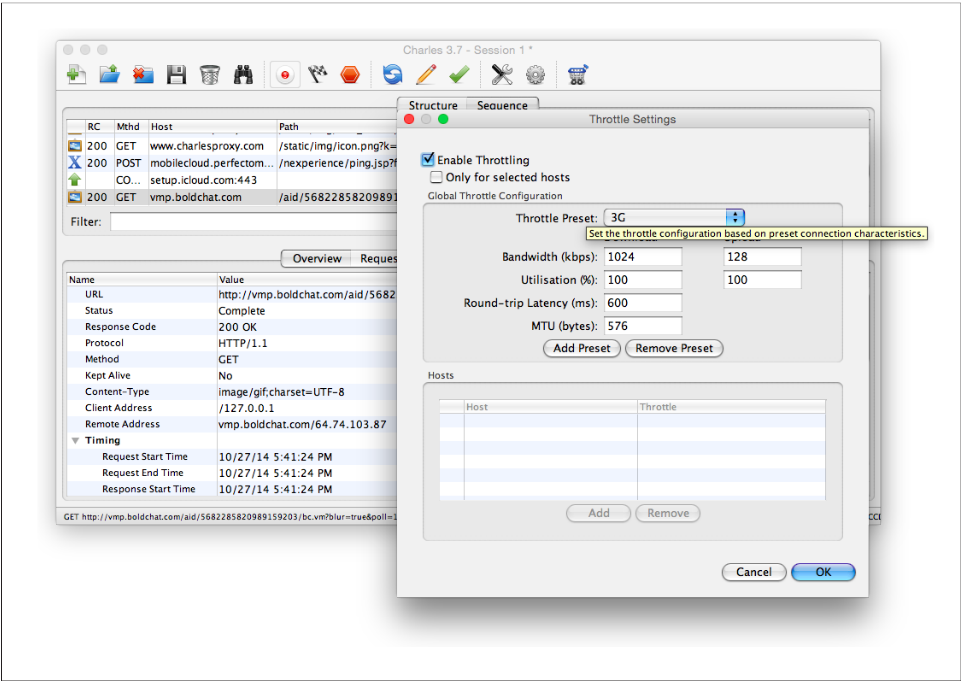
We can use it from a simulator, an emulator or even from a real device if we set a proxy on them.

Some tools available today are:

- Charles Web Debugging Proxy
- Mac Network Link Conditioner

- SlowyApp
- Clumsy
- NetLimiter
- Telerik Fiddler
- Chrome Network Conditioner

The Charles Web Debugging Proxy is an HTTP sniffing and proxy tool with a free version available that includes an HTTP throttling mechanism to simulate different bandwidths and latencies (as seen in Figure <<fig_charles_throttling>). It is available for Windows, Mac, and Linux.


Charles 3.7 - Session 1 *

	RC	Mthd	Host	Path
	200	GET	www.charlesproxy.com	/static/img/icon.png?k=
	200	POST	mobilecloud.perfectom...	/nexperience/ping.jsp?f
		CO...	setup.icloud.com:443	
	200	GET	vmp.boldchat.com	/aid/56822858209891

Filter:

Name	Value
URL	http://vmp.boldchat.com/aid/5682
Status	Complete
Response Code	200 OK
Protocol	HTTP/1.1
Method	GET
Kept Alive	No
Content-Type	image/gif;charset=UTF-8
Client Address	/127.0.0.1
Remote Address	vmp.boldchat.com/64.74.103.87
Timing	
Request Start Time	10/27/14 5:41:24 PM
Request End Time	10/27/14 5:41:24 PM
Response Start Time	10/27/14 5:41:24 PM

GET http://vmp.boldchat.com/aid/5682285820989159203/bc.vm?blur=true&poll=1

Throttle Settings

[✓] Enable Throttling
[ ] Only for selected hosts

Global Throttle Configuration

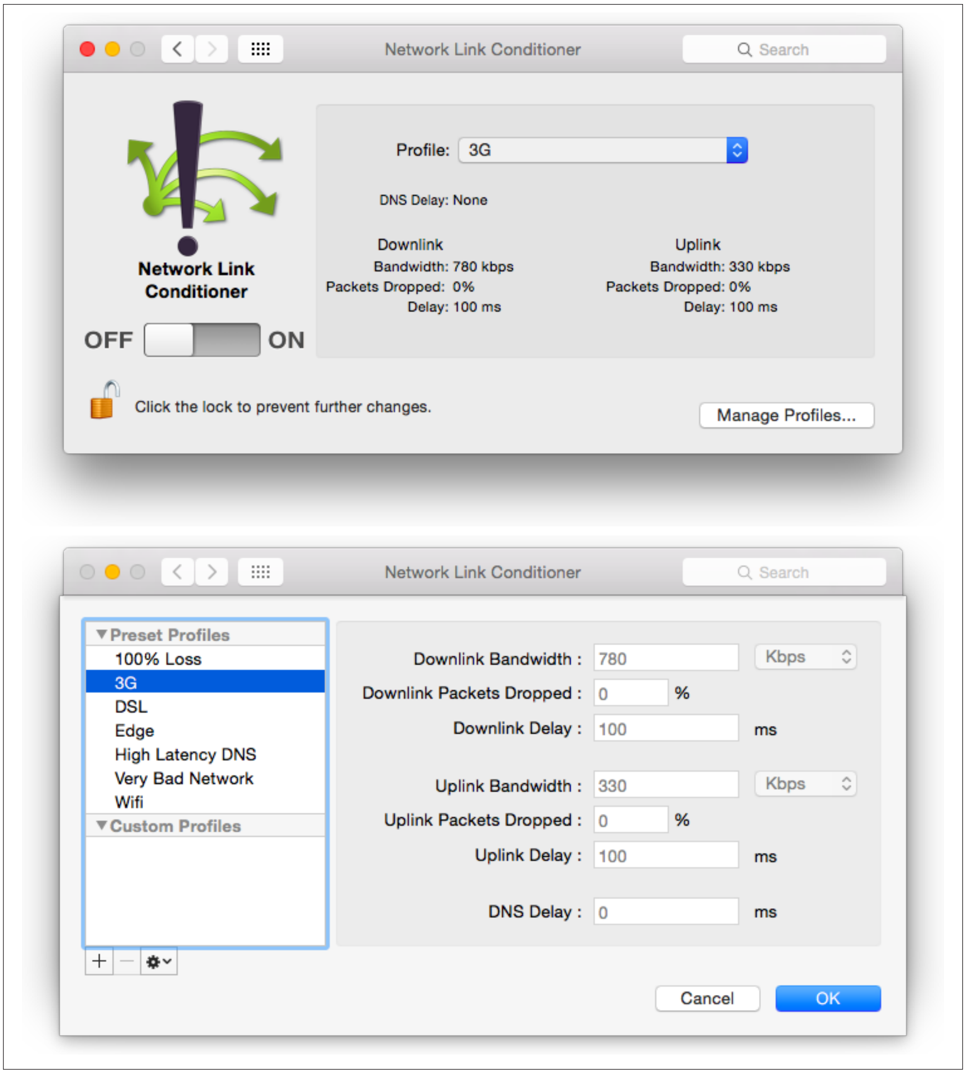
Throttle Preset: 3G

Set the throttle configuration based on preset connection characteristics.

Bandwidth (kbps):	1024	128
Utilisation (%):	100	100
Round-trip Latency (ms):	600	
MTU (bytes):	576	

Figure 2-16. With Charles Debugger we can easily throttle our connection and simulate a real cellular environment for emulators and real devices.

Network Link Conditioner is a free tool available as part of *Hardware IO Tools for Xcode* available only on Mac OS that will allow us to throttle the connection and emulate different conditions as you can see in Figure Figure 2-17. Once installed it will appear in the Mac Preferences app so we can easily change later.



Profile: 3G

DNS Delay: None

Downlink
Bandwidth: 780 kbps
Packets Dropped: 0%
Delay: 100 ms

Uplink
Bandwidth: 330 kbps
Packets Dropped: 0%
Delay: 100 ms

Click the lock to prevent further changes.

▾ Preset Profiles
- 100% Loss
- 3G
- DSL
- Edge
- High Latency DNS
- Very Bad Network
- Wifi

▾ Custom Profiles

Downlink Bandwidth :	780	Kbps
Downlink Packets Dropped :	0	%
Downlink Delay :	100	ms
Uplink Bandwidth :	330	Kbps
Uplink Packets Dropped :	0	%
Uplink Delay :	100	ms
DNS Delay :	0	ms

Figure 2-17. Once installed, the Network Link Conditioner will appear on your Mac's Settings menu and will let you simulate different real connections.

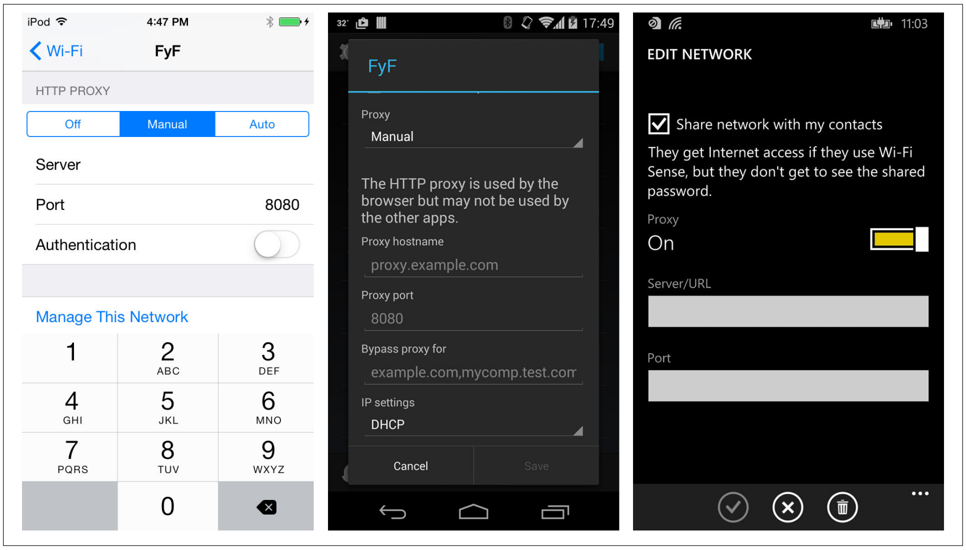
Slowy App is a Mac-only app that will simulate a real-world connection. It includes several presets, including bandwidth, packet loss rate, and packet latency. Slowy is not a proxy, so when using Mac-based simulators and emulators we don't need to configure any proxy.

Some of the connection simulators -such as Charles or Network Link Conditioner- allow us to create our own connection profile, setting bandwidth, latency and percentage of packet dropped. We can gather information from the market and then simulate different real scenarios.

Clumsy is a Windows-only app that will help us with the simulation of different connections, similar to the Mac utilities. NetLimiter is a Windows-only app for the same purpose. It includes a free limited version.

Fiddler Connection Simulator is a Fiddler proxy plug-in adding the ability to simulate different network connections on Windows computers. It now also support other platforms in experimental mode, such as Mac OS.

For the tools that are creating a proxy, we can then go to our mobile devices and set a proxy while connected to the same network as our proxy as you can see in figure Figure 2-18. Then we can see and throttle all the traffic from the mobile browser, web app or web view.

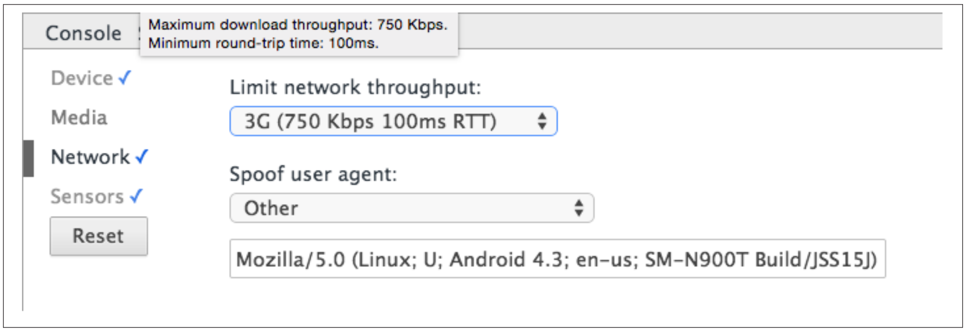

FyF
HTTP PROXY
Off [✓ Manual] Auto
Server
Port 8080
Authentication

FyF
Proxy: Manual
The HTTP proxy is used by the browser but may not be used by the other apps.
Proxy hostname: proxy.example.com
Proxy port: 8080
Bypass proxy for: example.com,mycomp.test.com
IP settings: DHCP

EDIT NETWORK
✓ Share network with my contacts
They get Internet access if they use Wi-Fi Sense, but they don't get to see the shared password.
Proxy: On
Server/URL
Port

Figure 2-18. If you have a desktop-based proxy you can set it from a mobile phone so the later will browser through that throttled connection

Finally and only available for Chrome, the Devices pane we covered before to simulate different mobile conditions includes a *Network Conditioning* solution. Network conditioning does bandwidth throttling and latency manipulation as you can see in Figure 2-19.


Console

Maximum download throughput: 750 Kbps.
Minimum round-trip time: 100ms.

Device ✓
Media
Network ✓
Sensors ✓
Reset

Limit network throughput:
3G (750 Kbps 100ms RTT)

Spoof user agent:
Other

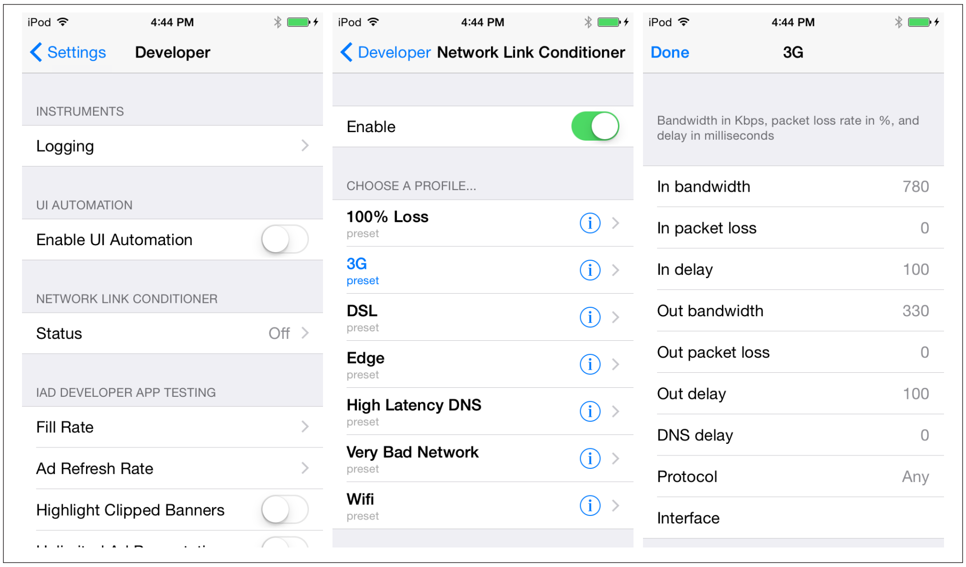
Mozilla/5.0 (Linux; U; Android 4.3; en-us; SM-N900T Build/JSS15J)

Figure 2-19. If you use Chrome Developer Tools with the Devices pane opened you can throttle the connection using the Network Conditioning ability.

On-device connection simulators

When we have a real device we can also simulate a different connection on-device.

iOS has a tool available already on every device from iOS 6. To enable the feature we need to set our device for development. We do that from Xcode, going to Window > Organizer, finding our device that should be connected to USB and selecting *Use for *development** as you can see in Figure Figure 2-20.


iPod 4:44 PM
Settings Developer

INSTRUMENTS
Logging

UI AUTOMATION
Enable UI Automation

NETWORK LINK CONDITIONER
Status Off

IAD DEVELOPER APP TESTING
Fill Rate
Ad Refresh Rate
Highlight Clipped Banners

iPod 4:44 PM
Developer Network Link Conditioner

Enable

CHOOSE A PROFILE...
100% Loss preset
3G preset
DSL preset
Edge preset
High Latency DNS preset
Very Bad Network preset
Wifi preset

iPod 4:44 PM
Done 3G

Bandwidth in Kbps, packet loss rate in %, and delay in milliseconds

In bandwidth	780
In packet loss	0
In delay	100
Out bandwidth	330
Out packet loss	0
Out delay	100
DNS delay	0
Protocol	Any
Interface	

Figure 2-20. After setting your phone in Developer mode, you will find the Network Link Conditioner option under settings to emulate a different connection over your phone.

After that you will find a new Settings options in your device. Open Settings > Developer > Network Link Conditioner to throttle your current connection based on a different profile.

> Besides throttling the connection on a real phone you can always force your cellular connection to work in a different mode, such as in 3G or 2G mode from the Settings menu. Just remember to go back to the default value after the testing (or you can keep it and feel the pain as any user in remote areas)

Android doesn't provide of a similar tool for on-device simulation. The only similar tool is available for the Android emulator and is known as *Android Speed Emulation*. It's a console-based tool when we open the emulator. For example, if we want to open an emulator simulating and 2.5G EDGE connection we can use

```
emulator -netspeed edge
```

By default it's emulating just bandwidth; if we want to also emulate latency on those networks, we need to use:

```
emulator -netspeed edge -netdelay edge
```

Possible accepted values are ***gprs***, ***edge***, ***umts*** or ***hsdpa***. You can also define your own customized values following the documentation

Picking the right target

As we saw during this chapter several options are available to help us test and measure performance, from real devices, simulators, emulators, remote labs and tools to make measurements accurate. Just remember that testing on a desktop is not enough to make conclusions about the performance of your web solution on a mobile device.

CHAPTER 3

Optimizing for the first visit

The first visit to our website is one of most important ones in terms of performance and the conversion impact of it. The user has just typed a URL, she has just clicked on a search result at Google or she just seen an interest tweet with a related link and she is expecting our content as soon as possible.

Failing in delivering the fastest experience in the first visit can lead to less conversion and people just leaving our websites for ever, or at least until the next opportunity. We know we don't want that and that's why we are going to cover the basics of optimizing web content for the first visit.

In this chapter we'll focus on the main techniques that will increase performance -and the perception of it- with normal effort. I'm saying this because later in this book we will enter into the extreme side of web performance that will make faster websites and better conversion but with bigger efforts in terms of implementation.

The HTTP side

HTTP -and HTTPS, the secure counterpart- is the protocol we are going to use for the initial load process when working with websites and webapps by default. This section is useless when talking about hybrid apps, as the initial loading will be based on a local filesystem as we will cover later in the book.

In this section of the chapter we will focus on tips that we can use on the server's configuration to improve performance.

In the HTTP section we will cover some server configuration tips. You should have some experience working with server's configuration (for example for Apache or IIS) as well as having permission to do so in your company.

If you don't have access you can always request to the server administrator to add these settings and in the case of shared hosting services, ask support or check in their web configuration panels.

We will cover some basic configuration steps for Apache, IIS and Express.js for Node.js. If you are using a different web server alternative, you will find instructions in the documentation.

HTTP Version

As a review, let's mention the HTTP versions that we have available today:

- HTTP/1.0, published in 1996 and still compatible with all the web browsers out there on mobile devices.
- HTTP/1.1, published in 1999 is the most used version on the web today.
- SPDY, a non-official open protocol based on HTTP/1.1 packages created by Google in 2012 and then used by many other providers to improve performance.
- HTTP/2, will be published in early 2015 based on SPDY and it will be the recommended version from now on being backward compatible with HTTP/1.1

There is no interesting reason to still serve websites using HTTP/1.0 by default for a mobile device today. A decade ago, some mobile browsers (the pre-iPhone era) were supporting 1.0 only but today you should verify that your server is supporting 1.1.

HTTP/1.0 was much simpler, the body is always plain and uncompressed and the TCP connection is always closed after sending the response, meaning that if the browser then needs to download more files to render the page it will need to start a new TCP connection to the server for that (involving several TCP packets back and forth).

HTTP/1.1 has several features that impacts positively the performance of your initial loading including tha abilities to compress the response's body and the ability to keep the connection alive waiting for more requests (usually stylesheets, scripts or images).

While HTTP/1.1 is probably the best option and every browser out there will support it, there are some proxies that might downgrade the TCP request to 1.0 so we should always have the option to support it.

Testing version support

To verify the version of your website, just make a cURL connection from the console to it asking for headers:

```
curl --head www.mobilexweb.com
```

By default cURL will try to make an HTTP/1.1 connection (on latest versions) so you can check the first line of the server's response to verify that, such as:

```
HTTP/1.1 200 OK
Date: Tue, 16 Dec 2014 21:58:16 GMT
Server: Apache
X-Pingback: http://www.mobilexweb.com/xmlrpc.php
Vary: Accept-Encoding
Content-Type: text/html; charset=UTF-8
```

If the first line says HTTP/1.0 it means you don't have the right configuration or you are browsing through a proxy that is downgrading your connections.

cURL: the quickest tool to debug HTTP connections

cURL is an open source project available for different platform that helps us making requests on different protocols including HTTP and HTTPS from the console.

It's available by default on Mac OS X and Linux. For Windows can be installed from the official website and it's also included in cygwin and Git Powershell.

There is also some online available solutions to use it quickly such as *http://onlinecurl.com*

If we want to make request in SPDY, there is an other project called gURL available at *https://github.com/mtourne/gurl*.

You can also force cURL to use one specific version using the following commands:

```
curl --http1.0 www.mobilexweb.com
curl --http1.1 www.mobilexweb.com
curl --http2 www.mobilexweb.com
```

Configuring HTTP/1.1

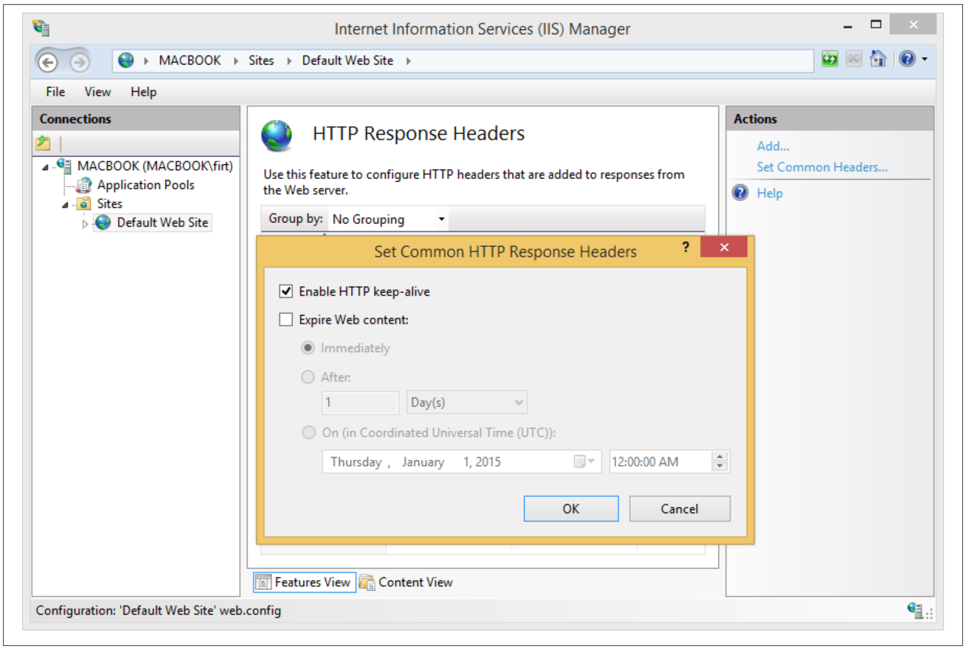
If you are running your server through Apache, you probably have support for 1.1. What you can define is the Keep Alive timeout, so the time that the server will wait for more requests before closing the connection.

You can then set the `KeepAlive` and `KeepAliveTimeout` properties on Apache's configuration both at the server's level (httpd.conf file) and at the virtual host's level.

For example:

```
KeepAlive On
KeepAliveTimeout 150
```

On Internet Information Server you can define these properties from the IIS Manager as you can see in Figure 3-1.



Use this feature to configure HTTP headers that are added to responses from the Web server.

Set Common HTTP Response Headers

[✓] Enable HTTP keep-alive
[ ] Expire Web content:
(●) Immediately
( ) After: 1 Day(s)
( ) On (in Coordinated Universal Time (UTC)): Thursday , January 1, 2015 12:00:00 AM

Configuration: 'Default Web Site' web.config

Figure 3-1. If you are using IIS, you can enable Keep Alive settings from HTTP 1.1 from the IIS Console per website.

If you are using Node.js and Express for your server, then Keep Alive will be enabled by default.

Adding support for HTTP/2

HTTP/2 will be the major next release and it will include several improvements for performance, such as:

- Multiplexed support (one connection for paralelism)
- Header compression
- The server can push responses into the client cache without waiting for an explicit request

At the time of this writing, HTTP/2 is not ready and only some experimental server implementations are available, including IIS in Windows 10 Technical Preview with a flag enabled.

While HTTP/2 gets a final version and browsers start to implement it, we can implement SPDY as we will see later in this section to get these advantages today.

A Node implementation of HTTP/2 is available at *https://github.com/molnarg/node-http2* following the standard http node extension API with addition for push.

Compression

HTTP/1.1 supports data compression, meaning that the browser will compress the file before sending it and the client must decompress it before rendering it. That means that on text-based files we can save up to 70% data transfer and we know that cellular connections are not reliable or fast, so saving data is a must.

Today, every mobile browser out there support the feature and we must have it enabled. And with current devices -even social phones-, there are no great drawbacks client-side on decompressing a file.

While there are some proxys and routers that might disable the feature most web servers will manage this automatically for us. Therefore it's safe to enable compression on text-based file.

The two common compression methods available are deflate and gzip. The gzip method includes a checksum for error checking that adds more CPU power both server and client-side. However, gzip is the recommended way on today's servers and clients. Server can be smart enough to pre-compress and cache hundreds of static files without the need of doing it per request. Dynamic-generated files will need a compression operation after the server finished the response.

Google has open sourced zopfli, a zlib compression library that can replace gzip and deflate with better compression results (around 5%) but much slower so it's preferred for static content that can be pre-compressed before serving.

Files to compress

While compressing files server-side seems like a good idea always, it might not be the case for every kind of file. All the text-based files involved in a website or webapp should be compressed, including: HTML, CSS, JavaScript, JSON, SVG and XML. In term of images only Icons (.ico) files worth compression and the old bmp format that nobody uses on the web.

Some mobile browsers today uses the website's icon (usually known as favicon) for the window's icon or for other UI purpose. Most websites uses the old .ico format for it and based on HTTP Archive data, an .ico file has an average 5Kb size uncompressed and compressing it cut that size in half.

However, have you ever zip a zip file? Once the file is already compressed you will probably don't save any space and that will happen to binary files that are already using some compression algorithms including image files, PDFs and audio.

In the case of font files (.ttf, .eot, .otf), while being mostly binary files they can usually take advantage of a gzip on top of them so you can also compress them. The only exception is Web Open Font Format (.woff files) that are already compressed.

Compressing very small files (let's say smaller than 800 bytes) won't make too much difference and in some cases (smaller than 150 bytes) will lead on a bigger transfer with the compression/decompression time. And when talking about small files that fit into a TCP packet there is no much difference in having it compressed.

The following table shows you the file formats that should be compressed:

Table 3-1. File Formats for compression in HTTP/1.1

file format	extensions	mime types
HTML	.html, .htm	text/html
XML	.xml	text/xml
Text	.txt	text/plain, text/text
CSS	.css	text/css
JavaScript	.js	text/javascript,application/javascript, application/x-javascript
JSON	.json,.js	application/json
EOT Font (Uncompressed)	.eot	application/vnd.ms-fontobject, font/eot
Open Font	.otf	application/x-font-opentype, font/opentype, font/otf
True Type Font	.ttf	application/x-font-truetype, application/x-font-ttf
SVG	.svg	image/svg+xml
Icon file	.ico	image/vnd.microsoft.icon

Enabling compression

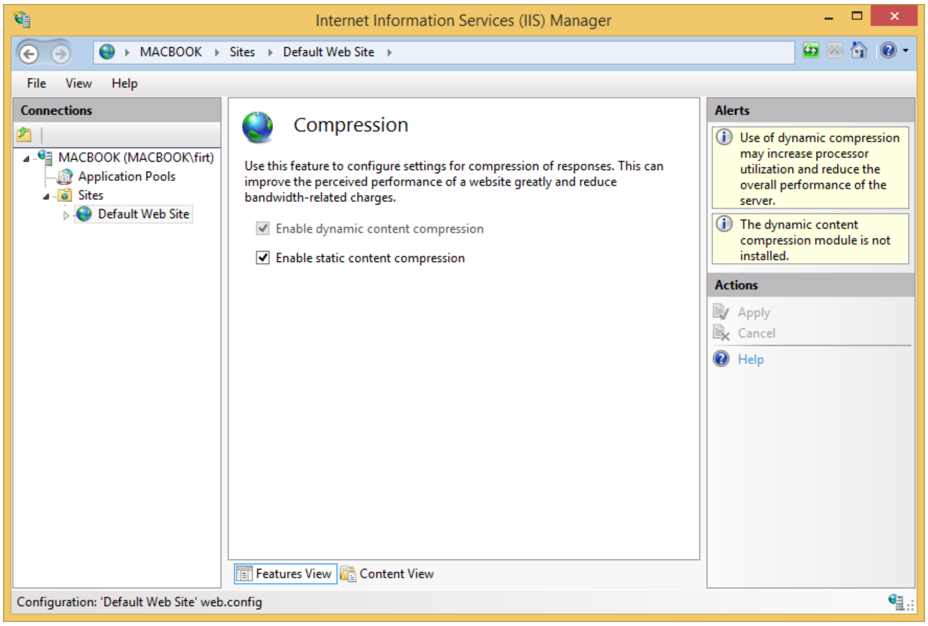
On Apache 2.x, you can set the compression through the mod_deflate extension. While the name says deflate it's actually using gzip; you can set the compression enabled based on the MIME type (the file format) using the following configuration declaration at the server level (httpd.conf) or at a folder level (.htaccess file):

```
AddOutputFilterByType DEFLATE text/html text/plain text/xml text/css text/javascript application/j
```

Setting compression by MIME type (using the Content-Type header used at the response) is better than by file extension (such as .html, .css) because when working with dynamic scripts (such as Python, PHP, .NET) the response is being generated by a script and not by a static file. The MIME type will always be there, both when a static file is being served and when a dynamic script is generating it.

There are also other independent extensions available for Apache that you can install and use.

On IIS, you can use the user interface at the IIS Manager to get into the Features View where you will find the Compression option. There, you will be able to setup the content compression for both static files and dynamic content as you can see in Figure 3-2.



Compression

Use this feature to configure settings for compression of responses. This can improve the perceived performance of a website greatly and reduce bandwidth-related charges.

☑ Enable dynamic content compression

☑ Enable static content compression

Use of dynamic compression may increase processor utilization and reduce the overall performance of the server.

The dynamic content compression module is not installed.

Configuration: 'Default Web Site' web.config

Figure 3-2. On Windows Phone, you can use IIS Manager to enable Compression

If you are using Node.js and Express for your web server, you can use the compression module as you can see at *https://github.com/expressjs/compression*.

While not always useful, you can also enable compression at the application level instead of the server level. For example, on PHP you can manage compression for the output with the `ob_gzhandler` function, for example executing as the first line:

```
ob_start("ob_gzhandler");
```

Redirections

Let me tell you a short real story. Let's say you want to access an airline's website and you type megaairlines.com (fake URL) on your tiny on-screen keyboard. The browser first will need to make a DNS request to get the IP address; when it's done, the browser starts the TCP connection and send a GET HTTP request. Then your server says: "oh, no, it's not here, you need to add www as a prefix" so your browser does that. It then makes other DNS request for www.megaairlines.com and again the TCP connections and the first HTTP request. Then your new browser says: "oh, you didn't tell me which page do you want to see, so you might want to go to the /HomePage".

Your browser, tired at this point, makes a new HTTP request to the www.megaairlines.com (thanks God we already have the IP address) asking for the resource "/HomePage". And then, the worst nightmare happens: the server realizes that you are on a mobile device and not on a desktop and it says: "Oh, I'm sorry you arrived here, because you need to go to m.megaairlines.com".

If you are still there, the browser (thirsty and hungry at this point) will make another DNS request to m.megaairlines.com and the TCP packets and the HTTP request and finally, the server says: "oh, you didn't tell me which page do you want to see, so you might want to go to the /HomePage". I'm glad that Artificial Intelligence didn't arrive to browsers yet because at this point it will want probably to kill itself. One more TCP connection and HTTP request and finally, finally, you've received your main HTML file for the airline (without CSS, JavaScript or Images of course).

Are you tired of reading this? Well, the same happens with the performance of your mobile website when you do this. And believe me, this is a real example and it was even worst than my story (I won't tell you which major airline is doing it).

Let's review the case when you access the website from a mobile phone:

```
1- megaairlines.com
2- www.megaairlines.com
3- www.megaairlines.com/HomePage
4- m.megaairlines.com
5- m.megaairlines.com/HomePage
```

So in this case they were 4 redirects, HTTP responses that says to the browser that "the resource is not here (temporarily or definitly)" so it should load it from somewhere else.

Every redirect will take from 100ms in the best case to 1 full second in the worst case. So in this example you could take up to 5 seconds for just nothing; the user will see a white screen for 5 seconds. It's the worst nightmare for the mobile performance as you can see in Figure 3-3.

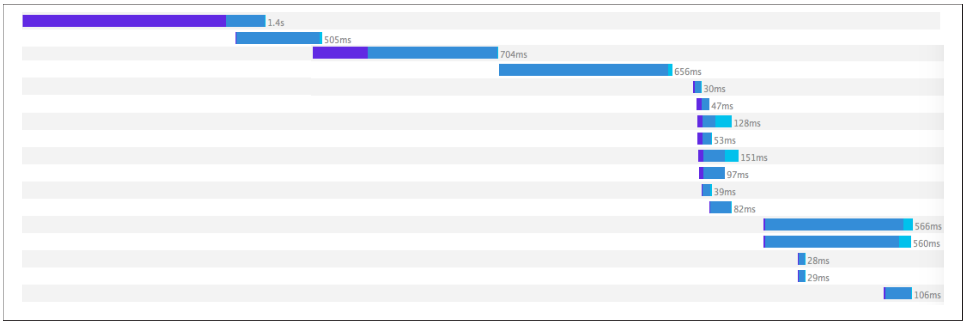


Figure 3-3. A waterfall chart when doing several redirects shows the impact in performance pushing every other measurement with no real advantage for the user

So HTTP redirects will harm your performance badly, so you need to reduce it at the minimum, being zero redirects the goal.

Sometimes we make redirects because we have different physical servers for the mobile version, sometimes because we didn't realize this was happening, sometimes because of a misconfigured web server. The reality is that today there is no real reason for HTTP redirects for the initial view.

Even if you have a weird architecture, you can always add a server-side layer that will make the translation for the user hiding the real architecture. And you need to count seconds as gold bar here, so avoiding redirects is a must.

I've heard some SEO experts saying that when you have a mobile version you must have a different domain (such as m.*). That's totally wrong, there is nothing wrong serving different HTMLs from the same URL based on the User Agent (the device) when you are talking about the same basic content. This was officially confirmed by Google.

Even if you still want to have a separate domain you can reduce the problem at the first load (for example, the home page). So the home page is being served from the main domain and all the links from that point will go to the mobile-specific domain. At least, you served the home page (initial view) as fast as possible and you keep your user engaged with your website.

If you can't avoid having redirects, at least reduce it to just one, having all the logic for getting to the final URL in one place.

To make things more complicated for us, there is now one or two redirects on a big amount of request coming to our websites that we can't manage or reduce: social networks and URL shorteners.

I'm sure you have seen URLs starting with g.co, t.co, bit.ly, tinyurl.com, etc. These services are basically HTTP redirects that we can't manage and will add up to one second delay on a mobile device when used.

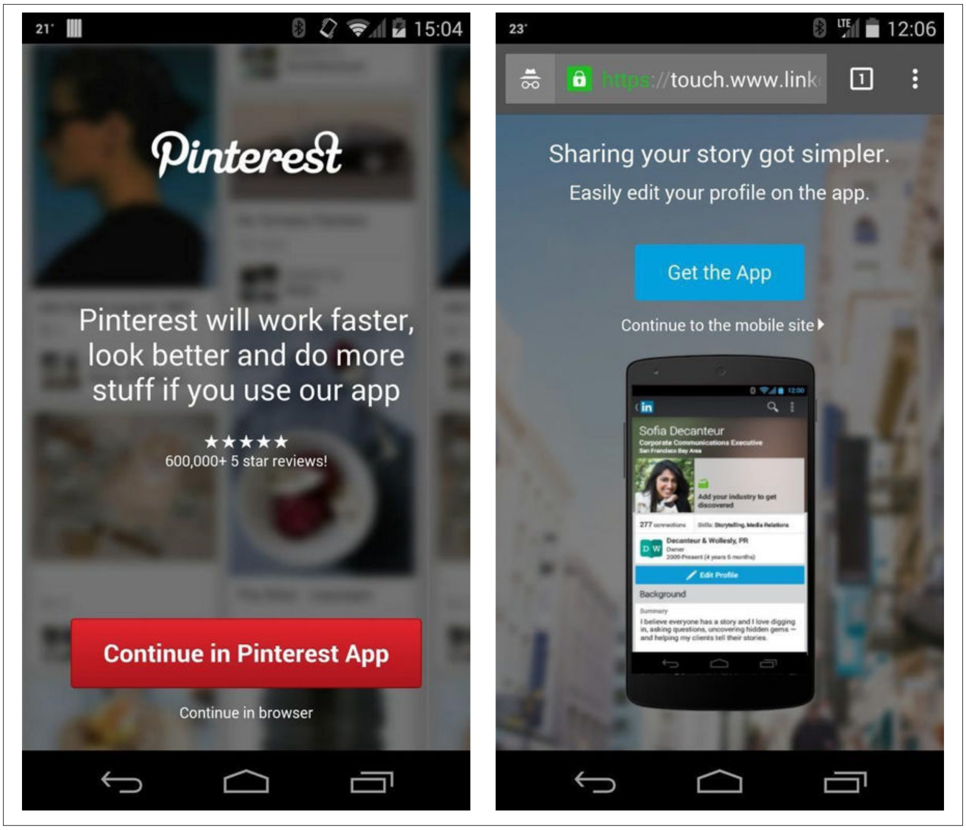
When you share a link on Facebook for example and the user clicks on it, it will also go through a Facebook redirect for stats purposes.

An HTTP redirect will need at least one HTTP request, on some situations a new TCP connection and on others, a DNS lookup. And we already know that latency on each packet on mobile connections can be up to 1 second on worst scenarios and around 200ms on average. Therefore, avoid HTTP redirects at all cost.

App (Spam) Banner

If we think on HTTP Redirects as "Slow down" signs, an App Banner is a "Stop" sign. An App Banner is a splash screen that acts as an intersticial HTML before the HTML the user was expecting from your server. These banners usually are promoting the native apps from your company, usually saying that the native apps offer a better experience.

I like to call them App Spam Banners; it's spam because the user didn't request it and you are stopping her to see the content you promised. Many big companies are using this technique without realizing how bad are they harming performance.



Pinterest will work faster, look better and do more stuff if you use our app

Sharing your story got simpler.

Easily edit your profile on the app.

Figure 3-4. An App Banner will stop the user from accessing the information she was expecting, therefore adding frustration to the webs' performance

Why are we harming performance? We need to remember that from a user's point of view, performance and fast has not always to do with the page load time. It's also about the perception of the whole load process. Therefore, if you first load a big image banner and wait for the user to process what you are asking and click the -usually- small link saying "Continue to the website", the final experience in terms of performance is terrible.

When you add an App Banner as a full page first response, you are pushing the page load time in at least 5 seconds.

The experience is even worst when you are adding this in your home page or entry point to your website (from a search engine search results for example). Think about this: you have just typed a URL for the first time or you clicked on a search result and the first thing you receive from the website is a banner asking you to install an app. You don't know who they are, you are not trusting their company yet, so why are you going to install an app? If you are brave enough, you will find the small "Continue to the website" link. If not, you will probably press the safe Back button and continue with another website.

Promoting your app smartly

There are two ways to promote your app without affecting the first visit performance and therefore, the first impression: 1) patience and 2) platform tools.

For patience I mean, wait for the user to trust you before promoting your app. A great example today is TripAdvisor.com. Trip Advisor gives you thousands of reviews from hotels, restaurants and touristic sites. Every time you search for a hotel or restaurant review in any city in the world, you will probably end on a Trip Advisor search result. When you access it from a mobile browser you won't see any App Banner until you've browsed to at least 5 different pages inside the site. If you've visited several pages it means that you are finding the website useful and you are starting to trust it, so getting an App Banner at this point (see figure Figure 3-5) won't harm it as if it were located in the first entry point.

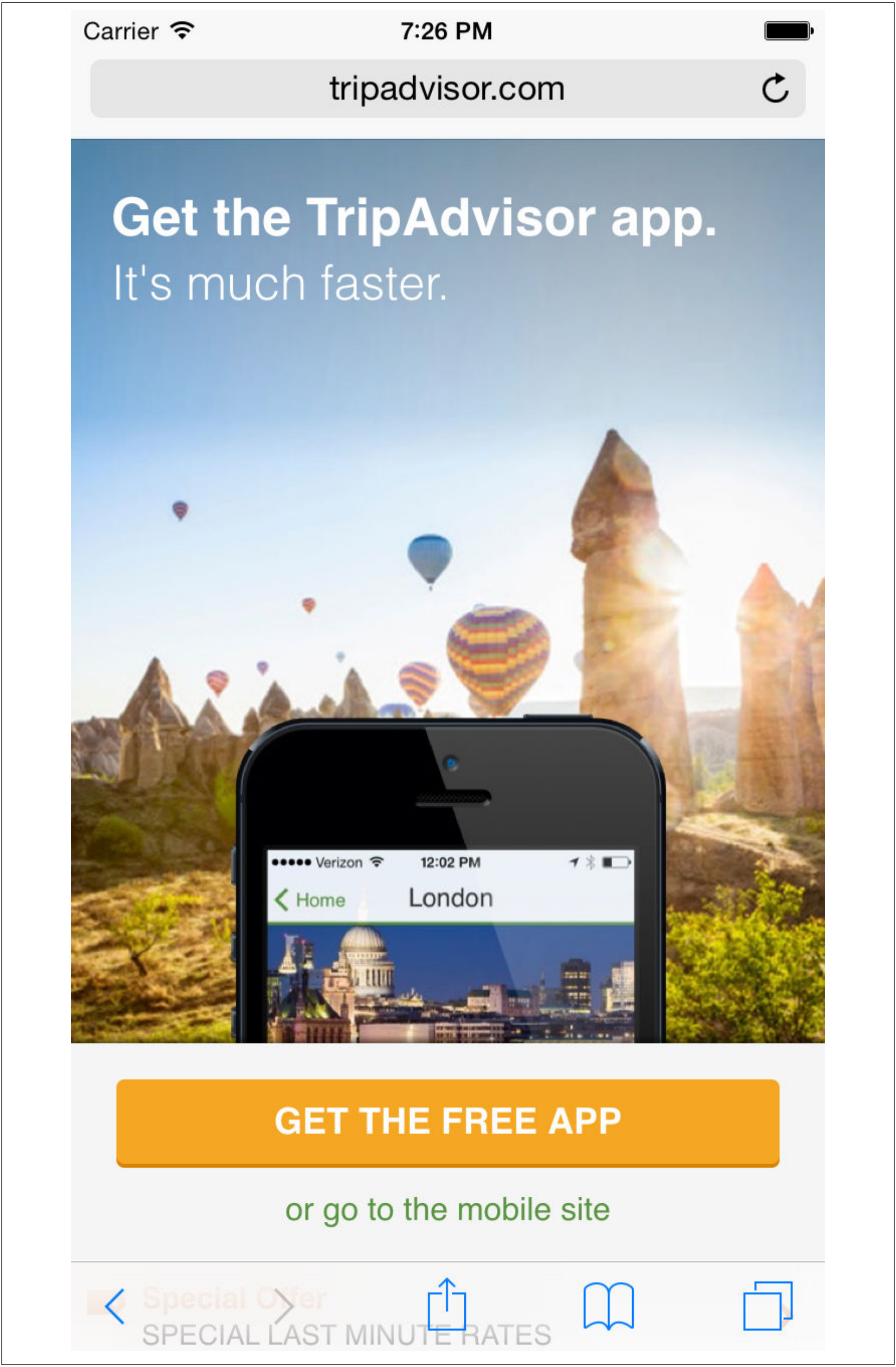


Figure 3-5. Trip Advisor waits four page views in the session before giving you an app

banner which improves the general perception of performance. Also, the app banner is not an interstitial page and it's included dynamically on the expected page.

The other solution is to use platform tools. At the time of this writing, Safari on iOS and Internet Explorer for Windows allow us to set some meta tags to promote your native app on your website. Instead of stopping the user to see the requested content, we are going to promote the native app in a non-blocking way (as seen in Figure Figure 3-6)

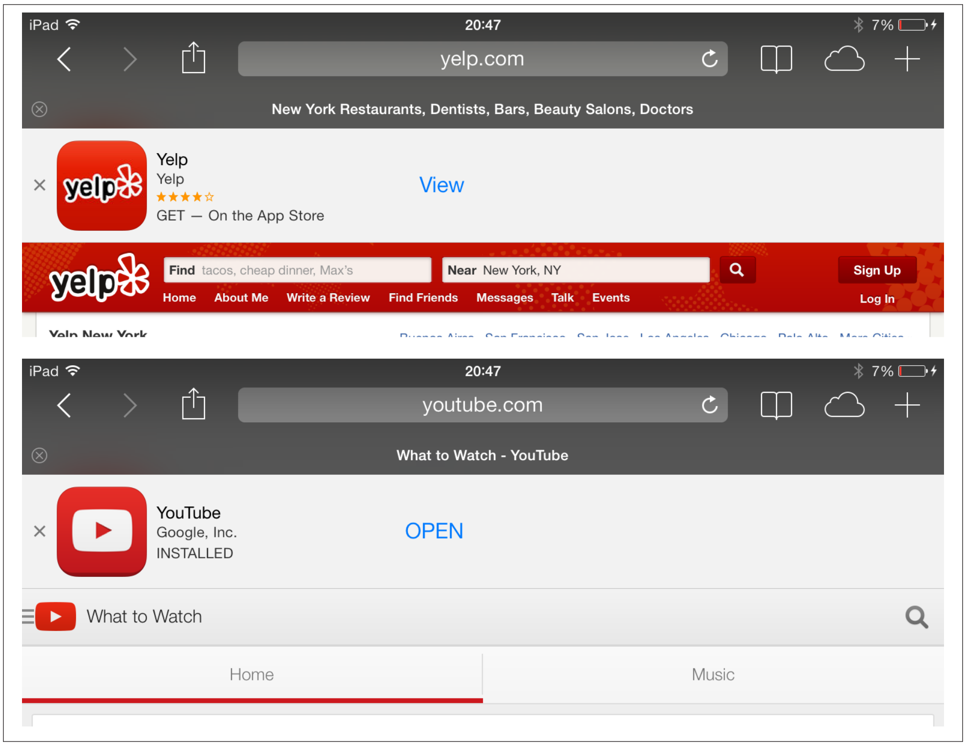
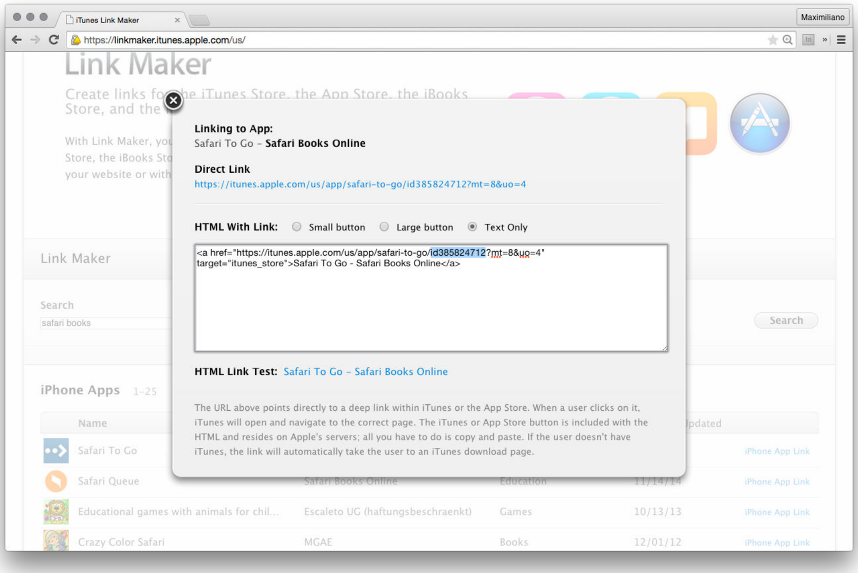


Figure 3-6. With Safari on iOS you can use an Smart App Banner that will suggest your native app in a non-intrusive way

iOS Smart App Banner

To define a Smart App Banner in an HTML document, we use a meta tag called `appleitunes-app` with a few declarations, including:

To make it work, the app you want to link to needs to be already approved on the Apple App Store. Then, you can go the iTunes Link Maker to retrieve the application ID you need to use in the meta tag. The ID is a nine-digit numeric value; to find it, search for your app by name on the Link Maker and extract the number from the URL, as seen in Figure Figure 3-7.


Linking to App:
Safari To Go – **Safari Books Online**

Direct Link
https://itunes.apple.com/us/app/safari-to-go/id385824712?mt=8&uo=4

HTML With Link: ○ Small button ○ Large button [✓ Text Only]

```
<a href="https://itunes.apple.com/us/app/safari-to-go/id385824712?mt=8&uo=4"
target="itunes_store">Safari To Go - Safari Books Online</a>
```

HTML Link Test: Safari To Go – Safari Books Online

The URL above points directly to a deep link within iTunes or the App Store. When a user clicks on it, iTunes will open and navigate to the correct page. The iTunes or App Store button is included with the HTML and resides on Apple's servers; all you have to do is copy and paste. If the user doesn't have iTunes, the link will automatically take the user to an iTunes download page.

Figure 3-7. To get the App ID for the Smart App Banner we need to make a search on iTunes Link Maker and copy the ID from the link

For example, to create a Smart App Banner for the app available for the iOS version of other of my books, we would use the following code:

```
<meta name="apple-itunes-app" content="app-id=393555188">
```

We can also send arguments (such as the current article or element) to the native apps through an app-argument inside the content.

Advanced topics for connecting your website to your native app is out of the scope of this book. You can learn more at the official docs of every browser or at my other book Programming the Mobile Web

Windows Store Apps

If you have a Windows Store app (from Windows 8 and Windows Phone 8.1), you can connect your website to it with some meta tags too without harming initial load's performance.

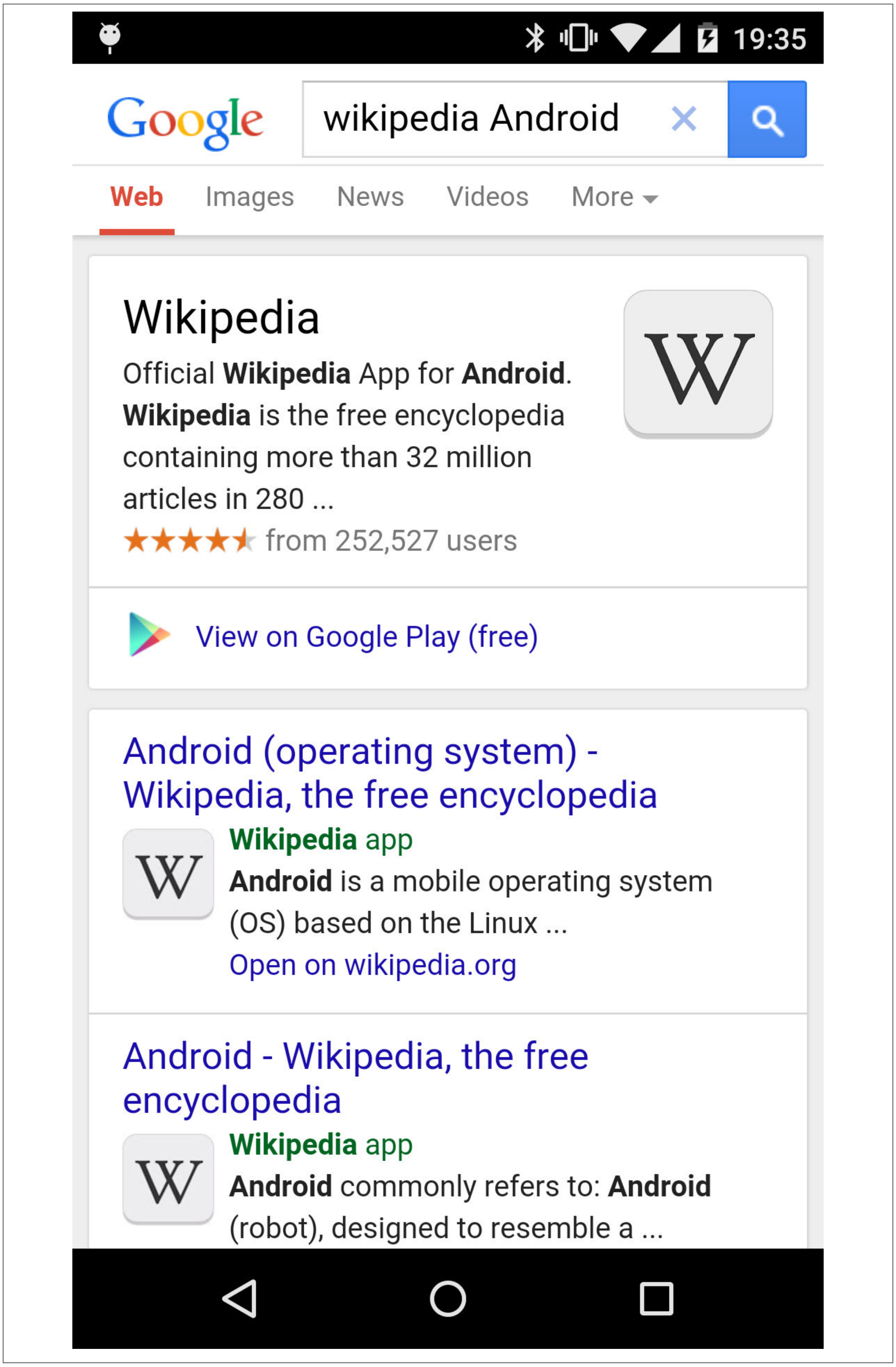
The two basics meta tags we need to define are `msApplication-id` with the identifier defined in the app's manifest file and `msApplication-PackageFamilyName` with the identifier created by Visual Studio to identify the app.

To get more detailed information visit the official documentation.

You can also link your Windows App to Bing's search results following the App Linking guides.

Android solutions

For Chrome on Android we don't have a similar solution yet, but we can use Mobile App Indexing for Google's search results. If you apply it, when the user is making a search and your website has a companion app, Google will promote it directly in the search engine as you can see in Figure Figure 3-8.



Wikipedia

Official **Wikipedia** App for **Android**. **Wikipedia** is the free encyclopedia containing more than 32 million articles in 280 ...

★★★★★ from 252,527 users

View on Google Play (free)

Android (operating system) - Wikipedia, the free encyclopedia

Wikipedia app

Android is a mobile operating system (OS) based on the Linux ...

Open on wikipedia.org

Android - Wikipedia, the free encyclopedia

Wikipedia app

Android commonly refers to: **Android** (robot), designed to resemble a ...

Figure 3-8. If you have an Android native app you can use App Indexing for Google's

search result. In this case on Wikipedia, you can open the native app directly.

On Firefox OS -and also Firefox for Android- we can detect if a webapp was installed using the Open Web Apps JavaScript API.

App Links

Finally, if you have a native app and you prefer your users to go there instead of the website you can also follow the open protocol AppLinks promoted by Facebook and used by other apps as well. If you set some meta tags on your website, compatible native apps (such as Facebook and Pinterest) will use them when your users share your content. Therefore, if a user is sharing a URL from your website, and another user is seeing it from a mobile device, these apps will try to honor your native link instead of the web link, opening your native app instead or inviting the user to download your app.

You can find all the meta tags in the AppLink Documentation website, but if you have your native app's ids from the stores this is how your meta tags will look like:

```
<meta property="al:ios:url" content="myapp://">
<meta property="al:ios:app_store_id" content="1234567">
<meta property="al:ios:app_name" content="My native app">
<meta property="al:android:url" content="myapp://">
<meta property="al:android:app_name" content="My native app">
<meta property="al:android:package" content="com.mobilexweb.myapp">
<meta property="al:windows_phone:url" content="myapp://">
<meta property="al:windows_phone:app_id" content="appid-from-store">
<meta property="al:windows_phone:app_name" content="My native app">
```

For App Links to work we need to set custom URI handlers (Intent Filters on Android) that will allow any app to open our app using a custom URI, such as myapp://

Requests

As we saw in previous chapters every request on a mobile website have a cost, involving optional DNS lookups, TCP packets, HTTP headers over unreliable and with high-latency connections. We've also seen that browsers will have a maximum amount of parallel request that can be done, pushing forward the request loading of other resources until some networks channels are freed up.

Therefore, we should reduce requests as much as we can being one request the best solution, but as you might expect not always the preferred one. We will get into extreme cases in following chapters, so for now, let's just try to reduce requests at the minimum and use some tricks to reduce the impact.

While in the following pages we'll cover techniques for reducing requests on CSS, JavaScript and images, let's first focus on reducing the impact of requests in the final loading experience.

Domain Sharding

We've already mentioned that most mobile browsers will download up to 6 resources at the same time from the same host. Therefore if your website needs 18 external resources, the download will be done in steps.

What happens if we split the resources in 2 or 3 different hosts? The browser's limits are per-host, and not per-IP address or per-server; therefore, it will be able to download 12 or 18 resources at the same.

The trick is known as Domain Sharding and it has the ability to impact on the final web performance when working with more than 15 resources when using 2 or 3 different domains in HTTP/1.1. Using more than 3 domains usually harms performance, as well as when your website uses just a few external resources.

If you are serving your website using SPDY or the next HTTP/2 Domain Sharding is not necessary and it will in fact harm performance because these new protocols allow concurrent downloads and request prioritization.

We are still using the same network channels and on some situations the same server, so there is a limit on the trick. However, because of the latency, making more requests at the same time allow us to reduce the final chart.

The impact of domain sharding will depend on the website but as a general rule, let's say that if you have more than 20 resources being currently downloaded from the same host and you are using HTTP/1.x Domain Sharding will improve your final performance.

The easiest way to implement Domain Sharding is to smartly decide after measuring performance, which resources are being queued because of the parallel download limit and move them to a different host. The new host can be a real new server or just a DNS alias (CNAME in DNS) to the same web server, for example mydomain.com can have an alias such as resources.mydomain.com or r1.mydomain.com and r2.mydomain.com.

As an example, if you have the following example running on mydomain.com:

```
<img src="images/img1.png">
<img src="images/img2.png">
<img src="images/img3.png">
```

```
<img src="images/img4.png">
<img src="images/img5.png">
<img src="images/img6.png">
<img src="images/img7.png">
<img src="images/img8.png">
<img src="images/img9.png">
<img src="images/img10.png">
<img src="images/img11.png">
<img src="images/img12.png">
```

It can be then optimized using:

```
<img src="images/img1.png">
<img src="images/img2.png">
<img src="images/img3.png">
<img src="images/img4.png">
<img src="images/img5.png">
<img src="images/img6.png">
<img src="http://r1.mydomain.com/images/img7.png">
<img src="http://r1.mydomain.com/images/img8.png">
<img src="http://r1.mydomain.com/images/img9.png">
<img src="http://r1.mydomain.com/images/img10.png">
<img src="http://r1.mydomain.com/images/img11.png">
<img src="http://r1.mydomain.com/images/img12.png">
```

In figure <<fig_sharding> you can see the differences in performance impact based on the waterfall charts.

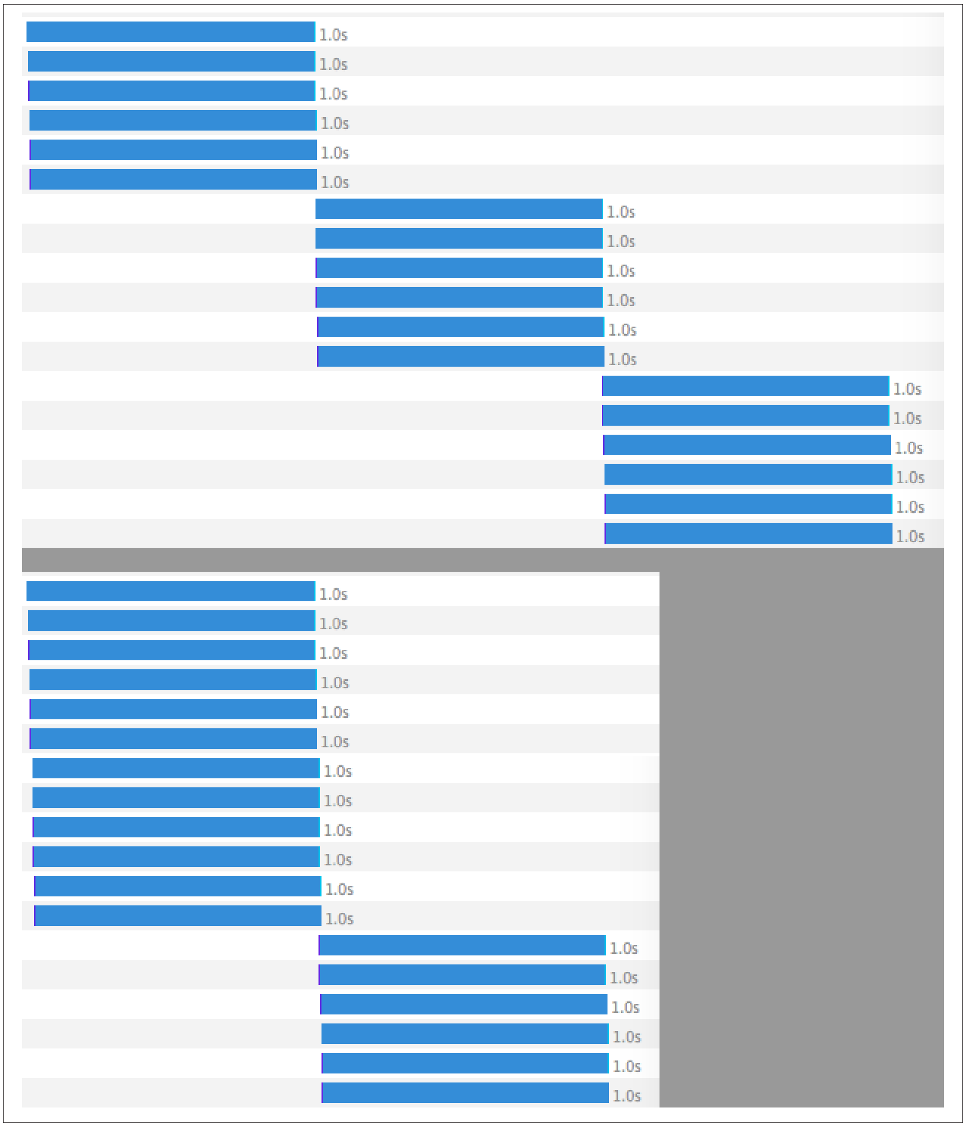


Figure 3-9. In this case, we see a waterfall chart without domain sharding (top) and the same website with domain sharding (bottom).

It's easy to think that we can shard resources across n domains then. However, tests show that sharding resources in just 2 domains will bring the best performance. Remember that the browser doesn't care about the final IP address, so if the two domains are being resolved to the same IP address the trick will work anyway.

You can say that our new solution has a bigger HTML, but remember we are gzipping our HTML, and the compression algorithm will take care of the repetitions. Therefore the bigger HTML might have a very low impact in the final compressed response through HTTP.

Cookie-less Domain

When you set a cookie from the client or the server, every request in the future to the same domain will send the cookie's data again. Every request. Therefore, if you have 30 external requests, you will duplicate 30 times the cookie's data in the upload streaming over cellular connection on some cases.

Cookies can have a size of 50 bytes to 4093 bytes based on what you are storing. A few years ago, the Yahoo! team did a research on cookie sizes and response times (over a desktop DSL connection) and they have found that for example a 1Kb cookie can impact around 100ms (per request with the cookie) and a 3Kb cookie can impact 150ms. Bandwith on mobile network might differ but in average it will be more than this.

Therefore, reducing cookies is a good idea. Mostly because they are being transferred on each request to the same server even if we are not really using them usually -such as when requesting images-.

To solve the problem we can:

- Reduce and/or compress cookies to the minimum
- Store the data server-side and use cookies to identify the record only
- Use cookie-less domains

Let's get deeper into the last option. If your additional resources are being served from a different domain that your HTML, then the browser won't send the cookies. Therefore, a cookie-less domain is another host -it can be an CNAME DNS alias to the same server- that will be used for linking external files.

Let's see a small example from mydomain.com with cookies:

```
<link rel="stylesheet" href="styles.css">
<script src="script.js"></script>
<img src="logo.png">
```

If our domain is setting a 1Kb cookie from the server, then the browser will upload that cookie three times to the server per request (with a potential penalty of more than 300ms). More important, we are not going to use that cookie server-side when requesting CSS, JavaScript or images.

Therefore, we can separate our resources to another host (let's say r.mydomain.com, r from resources):

```
<link rel="stylesheet" href="http://r.mydomain.com/styles.css">
<script src="http://r.mydomain.com/script.js"></script>
<img src="http://r.mydomain.com/logo.png">
```

The host r.mydomain.com can be the same server or a different one, but we set it up to not use any cookie at all.

When using a cookie-less domain we can also take advantage of domain sharding as we are using a separate host from the main HTML document

Freeing up our server

On some situations we can also use a different server for serving static files to free up our main server that sometimes is responsible for database access and scripting.

Using this technique also help us working with cookie-less domain and domain sharding.

We can do this in two ways:

- Using a light server for static files
- Using a CDN

A light server is usually a simple HTTP server, such as Lighttpd.

A CDN -Content Delivery Network- is a service provided by a company that will serve static files from us.

The advantages of using a CDN are:

- Great infraestructure and bandwidth
- High availability
- Mirrors located near the user -reducing latency-, but not as useful on mobile network as on desktop connections
- Automatic HTTP performant features on some providers, such as compression and caching

CDNs are typically commercial, including companies such as Akamai, Amazon and CloudFare.

HTML

We've already optimized the server to serve the first initial HTML as fast as possible, compressed and with a keep alive HTTP/1.1 connection.

Now let's see what can we do from the HTML side and the delivery of it to improve the initial loading performance.

Semantic Web

The first thing I have to say here is: create a simple, semantic, easy to read HTML code. I'm sure at this point I don't need to even remember you that you shouldn't layout your HTML using tables. But the new <table> is the abusse of <div>s and containers.

If you keep your HTML clean and semantic, using all the new HTML5 elements available, your HTML will be fast in transmission through the network, fast in parsing and fast in rendering, not even mentioning easier to understand when writing the CSS and JavaScript side.

Remember you are in a mobile device, do you really need all that CSS code for the first load? Don't get me wrong, I'm not saying that a mobile website should be a reduced or simple version, but thing about how important the first load experience is for conversion and reducing users abandoning your website without even seeing it.

Later in this book we will discuss how to deal with responsive websites, that is, websites that are serving the same HTML for desktop browsers, tablets and phones.

Flush the HTML early

We are going to see later in this chapter that on some situations we want the browser to start some processes as early as possible and by default, most web servers send the HTML response as one piece.

Let's say you have an HTML template that includes a header section as you are used to and the body needs to make some queries to a database that might take a while (let's say 200ms). Why can't we start sending some content to the browser before making those database queries? For example we can send to the server what we already know it will need, such as the URL of the CSS style to load or some information for DNS resolution.

We can do that flushing the response early through the HTTP channel. By default, most server-side script platforms such as PHP or ASP.NET are writing to a Buffer not to the real output when you are using writing functions, such as `echo`, `print` or `Response.Write`.

Most server-side scripts, have flush methods, including PHP, Python, Ruby, ASP.

When sending partial responses using a flush mechanism, a new HTTP header will be sent to the client `Transfer-Encoding: chunked`, so the client will know that the response will be sent in chunks.

However, if you have gzip enabled (and we know you should) some server-side scripts might buffer the output anyway before sending it to the client for gzipping it. So you should test the responses before applying it in production.

For example, in a simple PHP script we can flush early the header section of the HTML, before any database connection following this example:

```
<!doctype html>
<head>
 <meta charset=utf-8>
 <title>My super website</title>
 <link rel=stylesheet href=styles.css>
</head>
<body>
<img src="logo.png">
<?php flush(); ?>

<?php
   // Database access
?>
</body>
```

In the previous example we will send the response to the browser at least in two chunks, the head with the meta data information, the CSS file and the logo while the server will continue working on the database query to fill the rest of the HTML.

Therefore, the browser will anticipate the download of the external CSS and image and will probably render the logo faster compared to not flushing the document earlier.

DNS Prefetching

We already know that every time your website is using a new domain or host your phone needs to make a DNS lookup that on a cellular connection it might take up to 200ms. Every time you are adding a script or style from an external host (such as Facebook Like Button, Google Analytics Scripts, etc.) your browser will need to make a DNS lookup.

Previously in this chapter, we've also mentioned some techniques that will require the usage of other domains, such as Domain Sharding or Cookie-less domains. Therefore, if we can ask the browser to start a DNS lookup as soon as possible, we can gain some time.

Fortunately, some browsers support DNS prefetching, a way to start DNS lookups in advance that is really important on mobile websites. To do it we just need to set a <link> HTML element with a rel="dns-prefetch" attribute and the domain as the href attribute.

Because the DNS resolution has nothing to do with the http protocol, you can also set the href without it using just `//<domain>`.

For example:

```
<link rel="dns-prefetch" href="//google-analytics.com">
```

If we insert it at the top of the HTML and we flush it as soon as possible from the server, we'll perform better.

Some browsers have also a feature to prefetch DNSs from links (besides resources) that you can disable or enable based on your needs for next pages; we'll cover this feature in next chapters

DNS prefetching is working on Firefox, Chrome on Android, IE since Windows Phone 8 and Safari for iOS. If a browser doesn't support DNS prefetching it will just ignore the HTML declaration so its usage won't do any harm.

HTML Compression

There are some techniques in HTML5 that might reduce the size of your final HTML file, such as:

- Remove spaces, tab and new line characters
- Remove quotes on simple values, for example `<section id=news>` instead of `<section id="news">`
- Remove optional elements in HTML5, such as `<html>`, `<body>` and `<head>`
- Remote optional closing tags, such as `</body>`, `</p>`, `</li>`

While it's true that the final HTML size will be smaller we are not considering the HTTP compression that will apply on the server. With the compression in mind, the final file that we are going to serve has no real differences with the original HTML with spaces, new lines, tabs and all the elements.

Therefore, these tricks will make no big differences in performance. However, when you compress or minify your HTML and then apply HTTP compression you may have a small improvement. If you are using a publishing process -doing a set of things before publishing your website-, minifying the HTML won't make any harm and you might gain some milliseconds.

The compression tools that might have a bigger impact in performance are the ones that are doing a complex job with the HTML, CSS and JavaScript at the same time such as changing the names of ids and classes to shorter versions.

While compressing the HTML might not have a performant impact, you should remove HTML comments as a general rule as don't have any usage and they increase the final size, even compressed.

CSS

HTML5 doesn't work alone and it's very common to always have a companion CSS piece of code for styling and layout. Several questions arrive when working with CSS and performance that we will answer in this section.

Internal vs External

There are three ways to work with CSS: inline, internal stylesheets and external stylesheets.

Inline CSS involves the usage of the `style` attribute on one HTML element such as:

```
<section style="background: blue">
   Some content
</section>
```

Internal stylesheets are the ones that uses the `<style>` element with selectors, such as:

```
<style>
section { background: blue }
</style>
<section>
    Some content
</section>
```

And finally external stylesheets uses external files:

```
<link rel="stylesheet" href="styles.css">
<section>
    Some content
</section>
```

Unless you have some specific and rare situations, you don't want to use inline styling and compared to internal stylesheets it doesn't have any performance difference.

The question arrives when discussing external vs internal stylesheets. Before answering the question, let's remember two things:

- Each HTTP request has a cost in terms of TCP connections, HTTP headers and the high latency we have on mobile connections
- The browser will not start the rendering process until all the known CSS stylesheets are loaded

Based on the previous sentences, we can argue that internal stylesheets are better: they don't create a new HTTP request (they travel within the HTML) and the browser doesn't need to wait more time to start rendering.

Well, for the initial view internal CSS is the best. However, I know what you might think: what happens with caching and the rest of the HTMLs that may use the same CSS file. Well, here comes the part when you have to decide and find your own balance between performance for the initial view and performance for next visits.

What is clear is that for both internal or external stylesheets you need to avoid the inclusion of "future CSS", meaning CSS selectors that you are not really using in the first visit. It was common a couple of years ago to create just one big CSS stylesheet that is linked from every web page. It's fine from a cache point of view, but it will harm your user's first visit performance -on the most important visit-. Therefore, only use the CSS that you really need for that first view to reduce the rendering latency.

Only one external file

If you are going with an external stylesheet, be sure you will use just one file -so one HTTP request only-. You can join multiple CSS stylesheets manually into one file or you can use server-side scripts, tools and modules that will do it for you.

Loading External CSS

There are two ways to load external CSS files: through a `<link>` element -the HTML way- or through a `@import` declaration inside a stylesheet -the CSS way-:

```
<!-- The HTML way -->
<link rel="stylesheet" href="styles.css">

<!-- The CSS way -->
<style>
@import url('styles.css')
</style>
```

In terms of performance, the HTML way is the preferred one because the browser will realize it has to download a CSS while parsing the HTML code. On the CSS way the browser will realize it has to download the file after parsing the CSS that happens after the HTML parsing, so we are delaying a couple of milliseconds the request to the CSS file.

In terms of where to use the `link` element, the answer is: as soon as possible, and that is always at the top of your HTML. While technically you can insert references to external CSS files anywhere in the HTML page, doing it at the top will guarantee that the browser will download it and it will start the first render as early as possible.

Remember to flush early all your CSS declarations so the browser can start the download process as early as possible, that will lead to a faster rendering initial process and a better speed index.

Non-blocking CSS

As we already mentioned, the browser will block the rendering until all the known stylesheets are downloaded and parsed. Therefore, every CSS declaration you have - internal or external- is a potential performance issue that will affect initial rendering.

If you have a media query that is false at the time of the loading (let's say `orientation: landscape` when the browser is in portrait), the browser will download that file anyway but on some browsers at least it won't block the rendering process.

Even if you put your CSS link elements at the bottom of the page or with a false media query, the browser will prioritize it and download it before starting to render the page.

If you have some CSS stylesheets that you know for sure won't affect the initial rendering, you can use an asynchronous loading operation that won't push initial rendering.

At the time of this writing, the web community is working on a spec to allow a lazy loading for CSS external using a `lazyload` boolean attribute:

```
<link rel="stylesheet" lazyload href="external.css">
```

At the time of this writing, no browser is supporting this attribute so we need to hack it using a library of injecting a CSS using JavaScript after the rendering has happened.

Some libraries that can help you are loadCSS or AsyncLoader.

To inject it through JavaScript, we can use different JavaScript events, such as DOMContentLoaded, load or even a trick to detect the first paint operation (after the first rendering) using the Animation Timing API (known as `requestAnimationFrame`) on supported browsers.

For example:

```
// Multiplatform support
window.requestAnimationFrame = requestAnimationFrame || mozRequestAnimationFrame || webkitRequestA

if (window.requestAnimationFrame) {
        window.requestAnimationFrame(loadCSS);
} else {
    // API not available, we use the load
        window.addEventListener('load', loadCSS);
}

function loadCSS() {
        var link = document.createElement('link');
        link.rel = 'stylesheet';
        link.href = 'nonblocking.css';
        var head = document.getElementsByTagName('head')[0];
        head.appendChild(link)
}
```

Minifying

Even when doing HTTP compression, compressing or minifying the CSS will help in a smaller transfer and a faster rendering. There are plenty of tools available to the job including some online solutions, such as CSS Minifier or CSS Compressor.

For console tools, the YUI Compressor is a great tool while the YUI browser plugin already covered before will also do the work for you.

Web Fonts

Web fonts is the ability in CSS3 to load custom fonts to render on the screen. While it's a very nice feature, it has a big impact on performance.

Text by default is a non-blocking resource; that means that when you have text in your HTML, the browser will render it as soon as the render starts. However, when we apply a custom font, we convert a non-blocking text into blocking. As you can see in the next figure Figure 3-10, it's common to see empty boxes with no text on this situations while the font is being loaded.

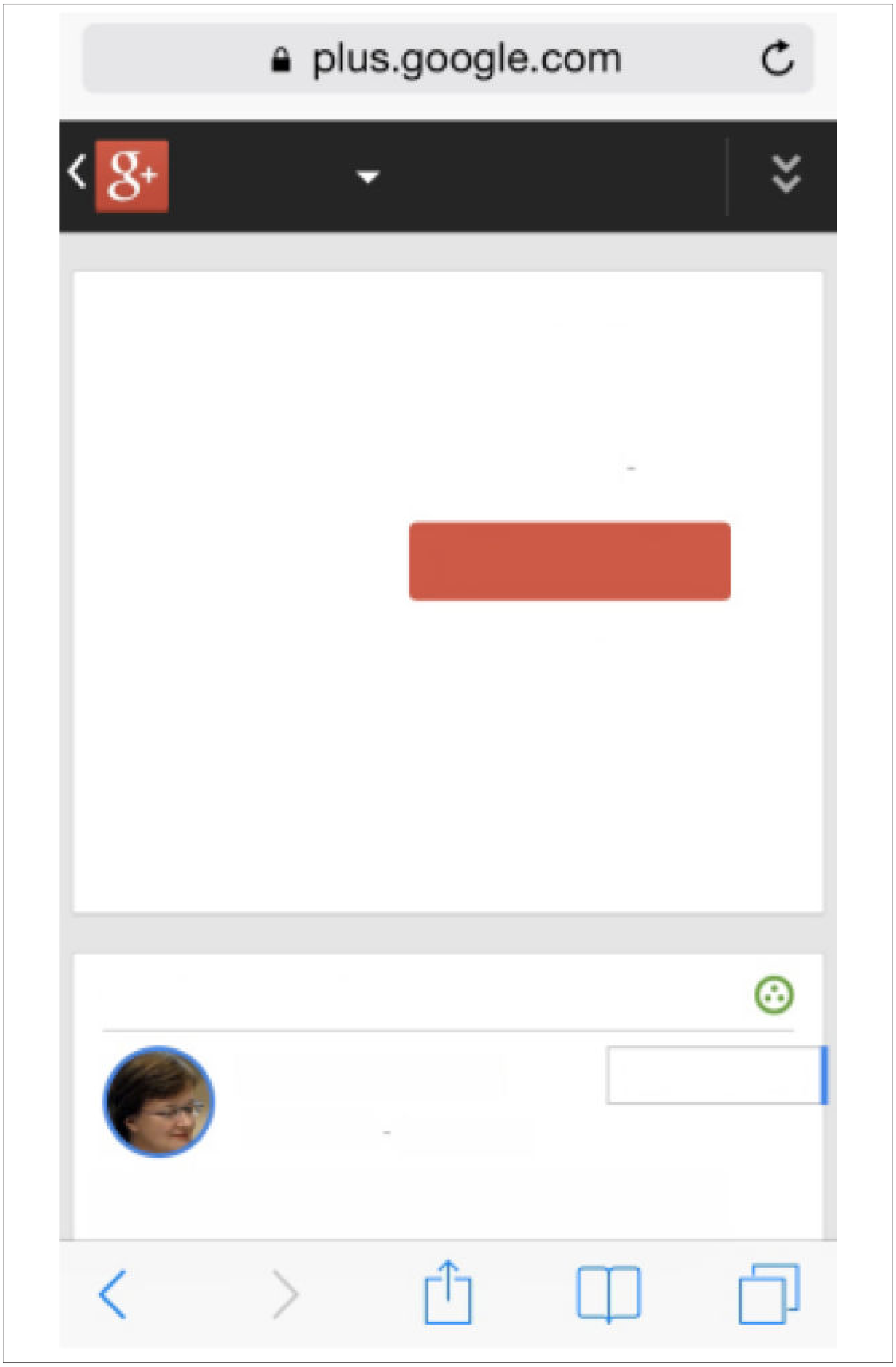


Figure 3-10. While a custom font is being loaded, some browsers will not render any

text at all.

Because of this problem, some mobile browsers (such as Chrome and Firefox) have started to define a request timeout in the last versions after enabling a default font family on the text. If after 3 seconds the font file wasn't loaded they will default to the fallback font. Safari on the other hand will hold the text rendering until the font file is complete while IE renders the text in the fallback font and then re-renders everything when the font is downloaded.

Every variant of the font (including italics, bold and bold+italics) will use a separate font file, so try to keep their usage to the minimum

We we mentioned before, besides WOFF format, we should compress all the other font file formats that will give us an additional 15% savings in data transfer.

Selecting Web Fonts

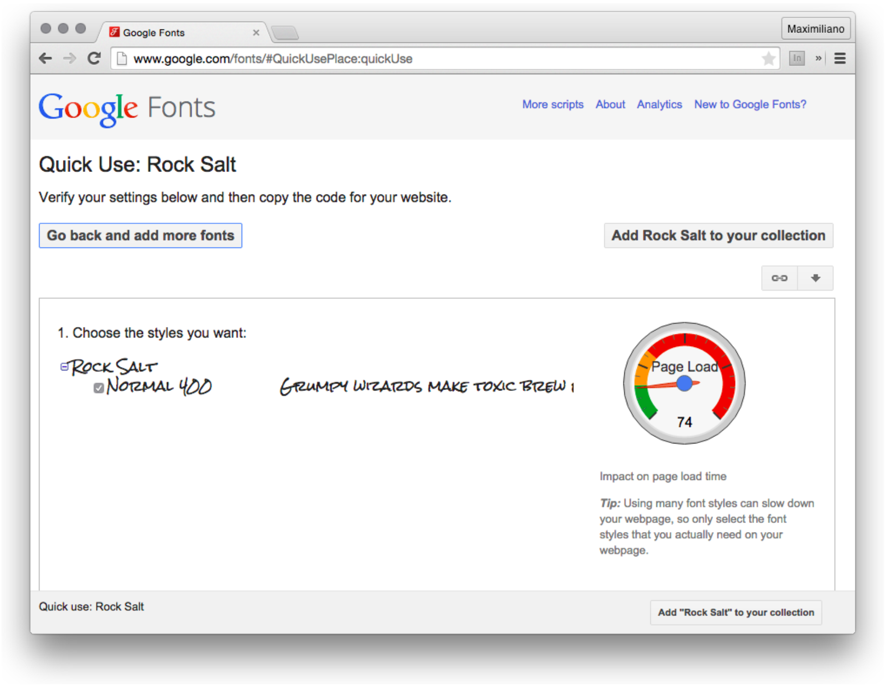
The browser needs to download the font before rendering the text; therefore the font file size should be as small as possible. Smaller files can be found when having simpler glyphs. Some font files are optimized for printing and therefore they have glyphs with more information that the one needed for a mobile screen. That means that there is room for optimization removing complexity from some glyphs on the font file when you have the right to do it.

Also, sometimes you can remove glyphs that you are not going to use at all, such as greek characters on a website primarily in English.

A free tool available on the web for font manipulation is Web Font Generator.

Web Font CDNs

Google has a free tool with more than 600 open source fonts available at Google Fonts. There you can pick one font, select the styles and character set we want and see the impact on the page load time for the selection as you can see in Figure Figure 3-11.


Google Fonts
www.google.com/fonts/#QuickUsePlace:quickUse
Maximiliano
More scripts
About
Analytics
New to Google Fonts?
Go back and add more fonts
Add Rock Salt to your collection
Page Load
74
Add "Rock Salt" to your collection

Quick Use: Rock Salt

Verify your settings below and then copy the code for your website.

1. Choose the styles you want:

Rock Salt
Normal 400 — Grumpy wizards make toxic brew!

Impact on page load time

Tip: Using many font styles can slow down your webpage, so only select the font styles that you actually need on your webpage.

Quick use: Rock Salt

Figure 3-11. At Google Fonts we can have a preview of how much impact that font will add to our website's performance

Adobe has also a selection of more than 500 Web fonts available to use at Adobe Edge Web Fonts.

The advantage to use a Font CDN, such as Adobe Edge Web Fonts or Google Fonts instead of serving the files ourselves is that the user might have already the font cached from other website using the same provider.

Loading Web Fonts

Web Fonts are declared in a @font-face declaration inside a stylesheet. Therefore if we want the loading process to be as fast as possible we need to make sure the browser read that declaration as early as possible. Therefore, internal stylesheets is the preferred way so the browser doesn't need to download first a CSS file to realice it needs to download a font file.

```
<style>
@font-face {
        font-family: MyFont;
        src: url(myfont.woff);
}
</style>
```

Other additional requirement for the browser to start a font download operation is to know that the font file will be used, so we need to have a CSS declaration using that font family also at the top. Some browsers (such as Chrome and Safari) will go further and they won't download the font file if you apply it with CSS on an empty element with no text.

When using Google Fonts, the fastest way to bring it without any JavaScript code is to use the link version; for example, to load the Open Sans font we can use:

```
<link href='http://fonts.googleapis.com/css?family=Open+Sans' rel='stylesheet'>
```

To have a finer control of font's loading, there is a new CSS Font Loading spec available on Opera and Chrome for Android only at the time of this writing.

Using the API we can request a font loading form JavaScript and detect when a font is ready and probably make a CSS swap from a fallback font to the new available font.

```
var myFont = new FontFace("MyFont", "url(myfont.woff)", {});
myFont.ready().then(function() {
    // The font is ready to use
});
myFont.load();
```

While we get support for all the browsers we can use some frameworks that will have some fallback mechanisms for font loading such as fontfaceonload. This framework will use the API when available and will use a different approach on non-compatible browsers.

JavaScript

JavaScript is a big enemy of initial loading performance by default. The main reason is that JavaScript blocks parsing by default. That means that the browser will stop parsing the HTML when it founds a JavaScript code (internal or external) and it won't continue until it's being downloaded and executed.

Internal vs External

Following the same idea as in CSS, if you have some JavaScript code that must be there for the initial rendering, it will be always better to have it in an internal script compared to an external script which will need an additional HTTP request.

Remember than on HTTP, it's better and much faster to download a bigger file and several smaller files because of the latency and HTTP overhead.

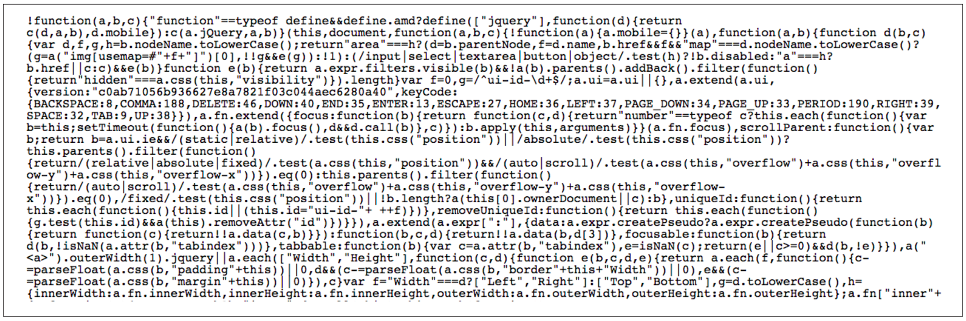
Therefore, if you have a script that is necessary for the initial rendering, inline it using a `<script>` element.

Minifying

JavaScript can be minified to reduce it's size and to obfuscate it. In terms of sizing, even when you are compressing your files on your server thanks to HTTP Compression you will gain around 15% when you are also minifying your code.

There are several minification tools available like the popular JSMin, the YUI Compressor or Google Closure Compiler.

The minified code will not have new lines, spaces and it might change the name of your variables, objects and functions based on the type of compression you are applying as you can see at the image Figure 3-12



```
!function(a,b,c){"function"==typeof define&&define.amd?define(["jquery"],function(d){return
c(d,a,b),d.mobile}):c(a.jQuery,a,b)}(this,document,function(a,b,c){!function(a){a.mobile={}}(a),function(a,b){function d(b,c)
{var d,f,g,h=b.nodeName.toLowerCase();return"area"===h?(d=b.parentNode,f=d.name,b.href&&f&&"map"===d.nodeName.toLowerCase()?
(g=a("img[usemap=#"+f+"]")[0],!!g&&e(g)):!1):(/input|select|textarea|button|object/.test(h)?!b.disabled:"a"===h?
b.href||c:c)&&e(b)}function e(b){return a.expr.filters.visible(b)&&!a(b).parents().addBack().filter(function()
{return"hidden"===a.css(this,"visibility")}).length}var f=0,g=/^ui-id-\d+$/;a.ui=a.ui||{},a.extend(a.ui,
{version:"c0ab71056b936627e8a7821f03c044aec6280a40",keyCode:
{BACKSPACE:8,COMMA:188,DELETE:46,DOWN:40,END:35,ENTER:13,ESCAPE:27,HOME:36,LEFT:37,PAGE_DOWN:34,PAGE_UP:33,PERIOD:190,RIGHT:39,
SPACE:32,TAB:9,UP:38}}),a.fn.extend({focus:function(b){return function(c,d){return"number"==typeof c?this.each(function(){var
b=this;setTimeout(function(){a(b).focus(),d&&d.call(b)},c)}):b.apply(this,arguments)}}(a.fn.focus),scrollParent:function(){var
b;return b=a.ui.ie&&/(static|relative)/.test(this.css("position"))||/absolute/.test(this.css("position"))?
this.parents().filter(function()
{return/(relative|absolute|fixed)/.test(a.css(this,"position"))&&/(auto|scroll)/.test(a.css(this,"overflow")+a.css(this,"overfl
ow-y")+a.css(this,"overflow-x"))}).eq(0):this.parents().filter(function()
{return/(auto|scroll)/.test(a.css(this,"overflow")+a.css(this,"overflow-y")+a.css(this,"overflow-
x"))}).eq(0),/fixed/.test(this.css("position"))||!b.length?a(this[0].ownerDocument||c):b},uniqueId:function(){return
this.each(function(){this.id||(this.id="ui-id-"+ ++f)})},removeUniqueId:function(){return this.each(function()
{g.test(this.id)&&a(this).removeAttr("id")})}}),a.extend(a.expr[":"],{data:a.expr.createPseudo?a.expr.createPseudo(function(b)
{return function(c){return!!a.data(c,b)}}):function(b,c,d){return!!a.data(b,d[3])},focusable:function(b){return
d(b,!isNaN(a.attr(b,"tabindex")))},tabbable:function(b){var c=a.attr(b,"tabindex"),e=isNaN(c);return(e||c>=0)&&d(b,!e)}}),a("
<a>").outerWidth(1).jquery||a.each(["Width","Height"],function(c,d){function e(b,c,d,e){return a.each(f,function(){c-
=parseFloat(a.css(b,"padding"+this))||0,d&&(c-=parseFloat(a.css(b,"border"+this+"Width"))||0),e&&(c-
=parseFloat(a.css(b,"margin"+this))||0)}),c}var f="Width"===d?["Left","Right"]:["Top","Bottom"],g=d.toLowerCase(),h=
{innerWidth:a.fn.innerWidth,innerHeight:a.fn.innerHeight,outerWidth:a.fn.outerWidth,outerHeight:a.fn.outerHeight};a.fn["inner"+
```

Figure 3-12. A typical JavaScript code after being minimized and obfuscated

Loading External JavaScript

To load external JavaScript we have only one standard way, a `<script>` element with a `src` declaration, as in:

```
<script src='jquery.js'></script>
```

The first question we may have is where to put these declarations: at the top or at the bottom?

If you have learnt HTML a couple of years (even decades) ago, the rule was to insert every `<script>` declaration at the top so the browser will download them first. It sounds good; until you realize you are harming performance, badly. So for years we've seen millions of websites with hundreds of script tags at the head blocking the parsing and leaving you with a white screen for seconds.

As a general rule, today the recommendation is to insert every standard `<script>` tag at the end, before the `</body>`. Therefore the browser will download every CSS and it will have all the content for start rendering the page before blocking the parsing because of the JavaScript download. We are removing the JavaScript code from the critical path.

When moving the JavaScript code to the bottom on a mobile connection there is a possibility that the user will engage with the content before the JavaScript was downloaded or executed so we should manage that situation.

defer attribute

If we don't need the JavaScript code to be executed before the page's load, then we can now use a boolean new `defer` attribute.So if we have this attribute, the page will download the file but executed it after the parsing has finished.

```
<script src='jquery.js' defer></script>
```

If you have several scripts with `defer` it's guaranteed that they will be executed in order.

Code should never contain references to `document.write` that writes in the HTML as an output from JavaScript when using this attribute.

async attribute

In newer browsers we can also specify that your script will not make changes on the HTML and it can be loaded asynchronously through the boolean `async` attribute.

```
<script src='jquery.js' async></script>
```

In this case the browser will not stop rendering when finding the script and it will download the file while it's parsing the rest and it will just halt parsing while executing it when it's ready.

default vs async vs defer

On mobile browsers, both async and defer are available since Safari on iOS 5.1 (every iOS device out there), Android Browser 3+, Chrome on Android, BlackBerry 7+ and IE on Windows Phone 8+. If the browser doesn't support them they will fallback to the standard behaviour.

The figure Figure 3-13 shows the difference between the default script, the async script and the defer script in terms of network, parsing and execution times.

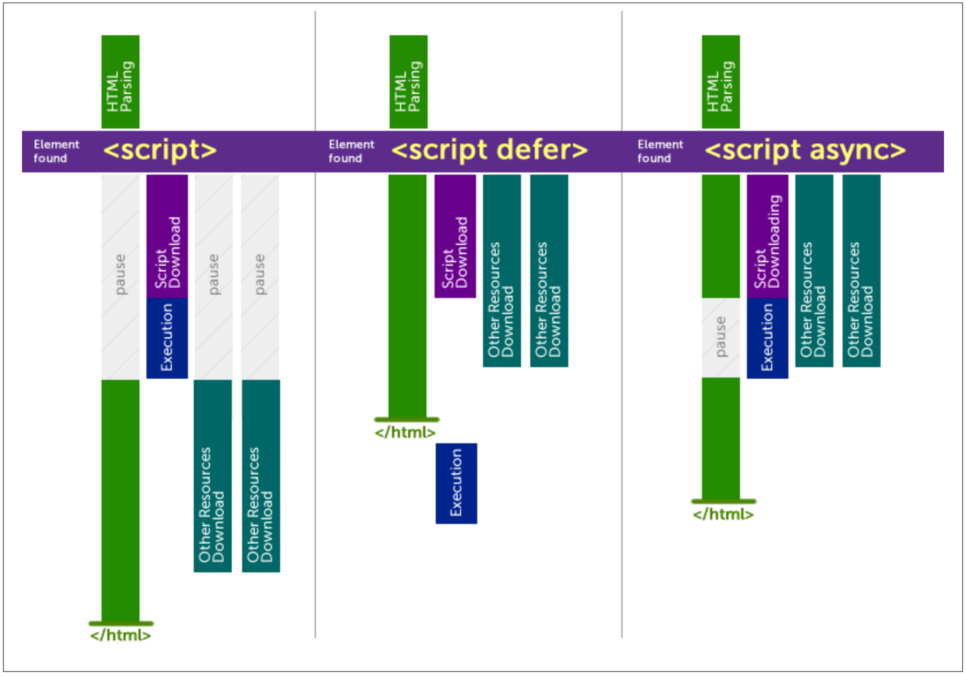


Figure 3-13. How network download, HTML parsing and JavaScript execution works with the three <script> tag's loading options available in HTML5.

If your script has a dependency, so it needs other JavaScript, you shouldn't use `async` as there is no guarantee in execution order when used. You can use `defer` in these cases.

In terms of performance, using `async` or `defer` will improve download and it will reduce the parsing block you have by default with JavaScript code.

Script loaders

Another way to load scripts asynchronously is to inject `<script>` tags with JavaScript instead of inserting it in the HTML directly. This idea lead to several loader frameworks available, such as RequireJS.

If you have any external widget in your website, including Facebook Like button, Twitter widget, Google Analytics you must use an async version of those scripts available on those websites to reduce the block parsing of your website.

Browsing like in mainland China

In 2014 I've travelled to China and I've dealt from dozens of websites suffering from what I know called the "mainland China effect".

In mainland China some websites, such as YouTube, Facebook and Twitter are blocked by the government -compared to Hong Kong and Macau, both in China but independent on some matters-.

What does this matter has to do with performance? Well, while visiting from my mobile phone several western websites -newspapers, blogs, and any sites- I had a very bad performance issue. Most of these websites were blocked for around 30/40 seconds while trying to load scripts from Facebook, Twitter or YouTube. And they were using a blocking standard script tag, so because those websites were blocked, the browser tries to download the script while blocking the parsing.

So, you might probably not know that you have this problem; and besides the chinese people browsing your website, what happens if one day Twitter's or Facebook's servers are down? Your website takes 40 seconds to render on users' screen when the timeout happens.

Therefore, to avoid the "mainland China effect" use always asynchronous scripts when using scripts from third-party servers. In fact, you can easily block these websites with a proxy or tampering with your hosts file to see if you suffer from this problem.

Using well-known frameworks

When using well-known frameworks -such as jQuery, Angular, Modernizr- you have the option to host the files yourself on your server or to link them from a CDN.

The advantage of linking them from a CDN is that there is a good chance that the user already has that file on the browser's cache thanks to a previous visit to a different website.

Some CDNs available for well-known frameworks include:

- Google Hosted Libraries
- Microsoft AJAX Content Delivery Network
- Bootstrap CDN
- jQuery CDN
- CDNJs

Only one external file

Following the same rule as in CSS, if you are loading external JavaScript files you should load only one (unless using a different asynchronous technique). That will improve the performance loading all the JavaScript code necessary in one shot reducing the latency in parsing.

Load Events

Most JavaScript developers are aware on the `load` event (and its HTML attribute `onload`). It's the second event you learn after the `click` when learning JavaScript.

The window's load event (used in `<body onload="">` in HTML) is executed when the whole page is loaded, and that means that the HTML and CSS was parsed and executed, the JavaScript was executed, the images were downloaded and everything is rendered on the screen.

If you are doing something after the onload that doesn't need all this (usually image loading), you can anticipate your code to the moment and increase user's perception of loading.

The new event available on all the mobile browsers today is known as `DOMContentLoaded`. This event will be fired when the DOM is ready (that is, the HTML is fully parsed and in memory) while the browser is still downloading resources. We can then bind event handlers, makes decisions and enable features before the onload. You can see the difference in action at Microsoft DOMContentLoaded Test.

If you are a jQuery developer, you've probably heard about the `$(document).ready` event that was a way to emulate DOMContentLoaded before it was available

Therefore, if you are currently using the `load` event, analyze what you are doing there and think about the possibility of moving it to DOMContentLoaded from the `document`, such as in:

```
document.addEventListener('DOMContentLoaded', function() {
        // Do something here
});
```

Images

As we mentioned before in this book, Images are non-blocking by default. When you have an image you are not blocking the parsing (as JavaScript does) and you are not blocking the rendering (as CSS does). The browser will leave a placeholder in the rendering while the image is being downloaded and decoded.

To improve rendering performance, we should define the image's dimension from the HTML or CSS side so the browser knows exactly the placeholder to use and it will reduce rerendering and repainting operations when the image is ready.

For example `<img src=logo.png>` can be improved using:

```
<img src="logo.png" width="300" height="100" alt="Logo">
```

Inline images

Inline images using Data URI is a way to embed an image in the HTML or CSS instead of requesting an additional file.

For example, we can define an img tag with the image itself inside it, without using an external file. This can be done using a base64 encoding of the image file—basically, storing the binary file as a set of visible ASCII characters in a string. This is great for small images, icons, backgrounds, separators, and anything else that doesn't merit a new request to the server.

> The size of an image (or any other binary file) will increase by about 30% when it's converted to a base64 string for a data URI, but its size will be reduced again if we are serving the document using GZIP from the server. Therefore, at the end it will be the same size or even smaller.

Some examples

The syntax is `data:[MIME-Type][;base64],data`. The data can contain spaces and newlines for readability purposes, but some browsers won't render it properly. It's better to maintain it in one line.

To convert an image file to a base64 string representation, we can use any online converter or command-line utility. There are free and online alternatives at Web Utils and Base64.

For example, the O'Reilly logo (original PNG file 75 pixels wide) attached as a data URI image looks like this:

*** <img width="100" height="17" alt="O'Reilly" src="data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAEYAAAARBAMAAAC-Si8f4AAAAA3NCSVQICAjb4U/gAAAAGFBMVEX///////8AAACpqanMzMxmZ-maHhoQ/Pz9kt3AEAAAACHRSTlMA/////////9XKVDIAAAAJcEhZcwAAC-xIAAAsSAdLdfvwAAAAcdEVYdFNvZnR3YXJlAEFkb2JlIEZpcmV3b3JrcyB-DUzQGstOgAAAAFnRFWHRDcmVhdGlvbiBUaW1lADEyLzExLzA5uegAp-**

**gAAAQNJREFUKJGVkUFTwyAQhfMXXiH1LA3hDMTeSVD-
PidW7WnMvkxn/vo+MsamX6s7wgOy3O29JgetR/
I2Rdy8B8HIcAyhj8PLIr0dsuYlDF8i8KWWC/BRKaVCUTaJmkhcDuNOHLV-
CaJg6VfJY6JivqGJ-
HEjn00q3tpsGcfd0KuuGdx2+fsmSl3m2r2k2gGg30i434xSMlmht0YbR
+EflAU71dMW89zzWcy2W61eF6YssK5j3vlII81Lj0vzOKHaX-
CuSz8334ybgC2bkdlUK6ZeMVvTdMM0M0IL3fmQskbD08LA8YHDzGDw9Nyn7Hziuv1hsH/
PltCQi9770MmsXFaGf/z3q/EFatlL/IFsBmgAAAAASUVORK5CYII=" /> ***

Performance Impact

While it seems like a good idea to reduce external requests, the problem with inlining images is that we are converting our by-default nont-blocking images to a blocking image.

If we inline the image in HTML then the browser will need to download and parse the image with the HTML, so it will be a bigger download and a bigger processing time before rendering.

If we inline the image in CSS, we are pushing the rendering time as CSS blocks rendering by default.

Therefore, I don't recommend using this technique as a general rule. However, there are some exceptions:

- When you are loading a CSS file asynchronously
- When used for really small images (such as bullets)
- When used for images that you really need for the first view, it's important as data and is what the user is expecting
- When there is no need to cache the image for future usage

As always the best suggestion on specific cases is to try two or three cases and measure them so you can make decision based on real data.

We are going to use this technique for other purposes as well later in this book.

Image Files

Even images being non-blocking, that doesn't mean that we shouldn't care about image size. The `load` event will not fire until all the images are loaded and that means on some browsers that the spinner or animation loader will not stop. And we've already mentioned that perception of speed is sometimes more important than the real speed.

Therefore, image load should be fast. To do that we need to compress our images and pick the right format per image. There is a lot of discussion around which format to pick and I will keep that outside of the scope of this book but I will suggest Smush It! as a great online tool for optimizing your images without loosing quality.

Optimization as a service

Some CDNs will offer you some optimization as a service solutions that will compress your images on the fly (sometimes with resize services). Services such as Sencha Src and WIT by Scientia Mobile

New optimized formats

Besides the well-known GIF, Animated GIF, JPEG and PNG formats, newer mobile browsers support new formats available that will help you in delivering smaller files with the same quality. Unfortunately, support for these new format is still browser-based so server-side adaptive delivery will be necessary:

- JPEG-XR, available only on IE is one evolution of the JPEG format that is optimized for high resolution images, creating better quality with less size compared to JPEG. It support both lossy and lossless pictures.
- WebP, open format created by Google, tries to replace both JPEG and PNG files (lossy and lossless formats) getting the same quality with a 25-40% reduction in file size. It's available on Chrome on Android, Opera and Android Browser since 4.1.
- Animated PNG (APNG), it's a non-standard animated format based on PNG that can create better and smaller animations compared to Animated GIF. It's available only on Safari on iOS since 8.0 at the time of this writing.

CSS Spriting

CSS sprites is a web design technique for reducing the number of image server requests on a web page. There are a lot of online resources and books available on this technique. For now, suffice it to say that if you have many images in your site (preferred logos, icons, background images, flags, and so on), you can reduce all of those to one big image with all the originals inside and use a CSS mask to determine which portion of it to show in each container.

This technique has a great impact on web performance. Instead of `<img>` elements, we will use any block element (div) or any block-converted element using display: block, such as a span or a tag.

Finally, in some browsers this technique can have an impact on rendering performance, because the big image will be duplicated in memory for each usage. We need to balance the performance gained through the reduction of requests with the performance lost in the rendering engine in some browsers.

So, if you have 10 icons each one on a different image, instead of 10 different requests (with all the latency involved per request) we can convert it to a 1 request only with all the 10 icons inside.

Conclusion

In this chapter we've covered basic tricks to improve the performance of initial view of your website or webapp without too much effort from our side. In the following chapters we will continue seeing techniques for improving future visits, the experience and perception after the page has been loaded and also how to apply other techniques that will allow us to get extreme performance on our mobile websites.

As a review, let's make a list of some important things we need to care about to improve performance:

- Server configuration, supporting HTTP/1.1 and compression
- Reducing requests and making resource loading cookie-less and with a possible domain sharding
- Optimizing the HTML for early flush and DNS cache
- Keeping CSS as short as possible, with fast loading to reduce render blocking
- Use new loading techniques for JavaScript to avoid parse blocking
- Optimize images to reduce network usage.