

Offline Apps, Storage, and Networks

For mobile browsers, HTML5 offers lots of JavaScript APIs that make our web architectures far more powerful than before. In this chapter, we will cover all the offline techniques, storage APIs, and network communication abilities that we have in modern HTML5 browsers.

Offline Web Apps

HTML5 allows us to create offline-capable web apps using a mechanism known as *application cache*. The concept is very simple. The user first opens the website in normal online mode, and it provides the browser with a package declaration text file called the *manifest file*, which lists all the resources (images, stylesheets, JavaScript, and so on) we want to be stored for offline navigation in the future. The next time the user visits the page, the HTML document is loaded from the cache, as well as all the resources in the manifest.

Our first step is to define what we want. Do we want a full offline application? Do we want some pages or data to be updated from the server every time the application tries to access them? Do we want to have a local data cache and update it whenever online access is available?

The second step is to define the manifest, or the list of files for the browser to download the first time the user accesses our website. This list must include every JavaScript script, stylesheet, image, or other resource that we want to access offline.



Firefox is the only mobile browser that asks the user's permission before storing an application cache (AppCache) package in the local memory. For all the other browsers it's just a transparent action from a user's point of view.

The architecture of our website will be exactly the same as if all the resources had been downloaded from the server. Images, stylesheets, and JavaScript scripts will be loaded, but they will be sourced from the cache instead of the server.

We can use application cache for:

Performance purposes

We can cache our home page and all its resources, resulting in faster load times and reducing the number of requests to the server. Every time the user accesses our URL it will load transparently from the local application cache and not from the network. The Google home page uses this technique on iOS and Android devices.

Installed apps

We can offer the user the option to install our app and add an icon to the Home screen, so it can be launched from there. Installed apps don't require a web server to work. *The Financial Times* web app (<http://app.ft.com>) is a good example of this idea.

An offline web experience

The website will be available even when the user doesn't have an Internet connection—we'll usually rely on other storage APIs for this. The Gmail web app for smartphones is a good example.

The Manifest File

The package list is delivered through a text file known as a *cache manifest*. This file must have as its first line the literal text `CACHE MANIFEST`. This line is followed by a list of all the URLs—relative or absolute—of the resources to download to the device. The manifest file must be served as `text/cache-manifest` and defined as the `manifest` attribute of the `html` element:

```
<html manifest="manifest.appcache">
```

The HTML file that is pointing to the manifest, as well as the manifest itself, will be stored locally in the application cache implicitly; we don't need to declare them inside.



If any file listed in the manifest fails to download while the package is being installed, the entire package is invalidated. That means that if we are defining resources on third-party servers, we will rely on these servers for our apps to be installed.

The initial `CACHE MANIFEST` line can be followed by a series of relative or absolute URLs that we want to be cached for offline availability. We can comment lines by using a hash (`#`) at the beginning of the line:

```
CACHE MANIFEST
# This is a comment
ourscript.js
images/logo.gif
images/other_image.jpg
ourstyles.css
```



While there is no standard definition, the manifest is recommended to use the *.manifest* or *.appcache* extension and must be delivered with the right MIME type: `text/cache-manifest`.

Reusing the manifest

If more than one HTML page that the user browses points to the same manifest URI, the manifest package will be reused and all the HTML files browsed will be added implicitly to the manifest as “master resources.” That is, after a package has been installed, if the user browses to another page that points to the same manifest, all the resources will already be available offline.

Accessing Online Resources

If our application attempts to access any resource that was not originally defined in the manifest file, the process will fail because the application is sandboxed offline. By default, all the resources are declared in an implicit `CACHE:` section. If we know for sure that we are going to need some information from the Web, we can define it in the manifest file in a special section called `NETWORK:`. (To define a section, we just end the line with a colon.) So, if we want a *countries.json* file to always be delivered from the server, we can change our manifest to:

```
CACHE MANIFEST

# Resources that should be installed on the user's device
CACHE:
ourscript.js
images/logo.gif
images/other_image.jpg
ourstyles.css

# Resources that should always be downloaded from the Web
NETWORK:
countries.json
```

Then, *countries.json* will not be downloaded with the other resources and instead will be accessed online every time the application needs it. If there is no Internet connection, we will not get this file, unless the browser has a cached version (using the typical web cache, not the application cache).

In the network section, we can use wildcards, such as *, or folders; every resource in that folder will be accessible from the Web while we are in an offline operation mode. So, if we want to have an offline application that can access the full Web if the user has a connection, we can just use:

```
NETWORK:  
*
```

With this configuration only files listed before the **NETWORK:** section will be loaded from the offline package; every other resource will be loaded from the Web.

Fallbacks

Application cache has a mechanism to access external online resources but at the same time provide a fallback to avoid errors if the user is offline: the **FALLBACK:** group. In this section we provide two URLs, space-separated; the first one can use wildcards while the second one should be a specific URL. For example:

```
FALLBACK:  
images/profile.png noconnection.png
```

In this example, when the web app tries to get *images/profile.png*, it will bring it from the server if the user is online and will return *noconnection.png* as a fallback file if the user is offline. The fallback image is stored in the application cache package implicitly.

Updating the Package

We've said that when the package is installed all the resources—including the main HTML document—will always be loaded from the local storage and not from the Web. Therefore, it's fair to ask: how can we update a resource? What happens if we want to update the theme CSS file, change an image, or add a new page link in the HTML document?

One part of the picture that I omitted earlier is that every time the user opens a cached web app, while the app is always loaded from the local storage, in the background the browser tries to get an updated manifest file from the server.

If there is no Internet connection, nothing happens, and the local version is used. If there is a network connection available, the browser downloads the manifest file from the server and does a byte-by-byte comparison of the new version of the file with the local version from the original web app download. If even one byte has changed, the entire cached manifest is invalidated and every resource is downloaded again, using the new manifest file.



We can force the browser to check for a new manifest file while the application is running, rather than waiting for the next time the app is launched, using `applicationCache.update()`.

Go back to the previous paragraph and read it again. Done? OK, let's take another look at what's happening. If we change the contents of a CSS file but do not change the name of that file, the manifest will be the same, so the downloaded files will not be updated. This is an important point—we need to change the manifest itself for the web app to receive the update.

How should we change the manifest when we make an update to any of the listed resources? The change can involve something as simple as adding a space, changing the resource name (versioning it), or even including a comment line at the start of the manifest file containing a random value or the last-modified date, as in:

```
CACHE MANIFEST
# webapp updated 2013-10-01
```

If we make a single change—for example, the date—the whole manifest will be invalidated and the platform will download all the files again. (Yes, *all* the files—with this API we can't update just one resource.)



Remember that the resources in the manifest will not be downloaded again until we update the manifest file or invalidate the `AppCache`.

There's another unpleasant problem that arises when dealing with manifest updates: if there is an update, the platform downloads all the resources in the manifest again, but this download process is done *in the background while the previous files are on the screen*. This means the user will not see the newly downloaded versions of the resources until she reloads the application.

In other words, if we change the manifest, the user must load the page twice to see the new version. As we'll see shortly, we can use events to handle this situation.

Deleting the Package

How do you remove a package from the application cache? The answer is not so obvious. The only way to do it is to deliver a 404 HTTP code when the browser is downloading the manifest while trying to check for an update. Therefore, if we want to delete a package we need to use Ajax (or some other technique) to communicate to the server that we want the package to be removed on the next reload. The server may physically delete the file, or just store a flag in some sort of server-side storage (such as a session variable)

so that on the next manifest request for that user we force a 404 status. After that, we can reload the app or just force an update using `applicationCache.update()`.



In some browsers, the user can delete the packages from an advanced configuration section, usually called “website data.”

The JavaScript API

There is a global JavaScript object that helps us to know the status of the application cache. The object is `applicationCache`, and it has a `status` property that can have one of the values listed in Table 13-1.

Table 13-1. Status of the applicationCache object

Value	Constant	Description
0	UNCACHED	This is the first load of the page, or no manifest file is available.
1	IDLE	The cache is idle.
2	CHECKING	The local manifest file is being checked against the server's manifest file.
3	DOWNLOADING	The resources are being downloaded.
4	UPDATEREADY	The cache is ready.

If the application cache status is 0, our document is loaded from the network; otherwise, it is loaded from the application cache.

We can use constants to ask about the current status. For example:

```
if (window.applicationCache!=undefined) {  
    // The API is available  
    if (applicationCache.status==applicationCache.UPDATEREADY) {  
        // There is an update waiting for reload  
    }  
}
```

The `applicationCache` object has the methods `update()`, which will force an update check, and `swapCache()`, which will swap from the older cached versions of the resources to the newly downloaded versions (if the object is in the `UPDATEREADY` state). However, the HTML document and all resources already in memory will not be updated until we do a full page reload (such as with `location.reload()`).



If your offline application needs to store custom images that are only for one user (for example, pictures of the user's contacts), you can create a manifest file dynamically for each user or, even better, store the images in base64 in offline storage for usage as inline images later.

Cache events

The `applicationCache` object has events that we can handle to manage every situation. For example, if the user is accessing our website for the first time, we can show a “Downloading app” message while the resources are downloaded so the user will wait, increasing the probability of a complete download.



We can use the new events `online` and `offline` and the `navigator.onLine` property to detect whether the user has an Internet connection available or not.

The possible events that we can bind to are listed in Table 13-2.

Table 13-2. Events available for `applicationCache`

Event property	Description
checking	The browser is checking the manifest.
downloading	The browser has started downloading the resources listed in the manifest.
progress	A resource has been downloaded (this event is fired for every resource, so we can create a progress bar). The current HTML page and the manifest file count as resources, so this event will be fired $n+2$ times, with n being the number of files in the <code>CACHE :</code> and <code>FALLBACK :</code> sections.
cached	The first download process has finished properly.
noupdate	The cached manifest has been compared with the version on the server, and no update is available.
updateready	There was an update, and the new resources have been downloaded properly and are waiting for a reload.
error	There was an error downloading a resource.
obsolete	When checking for an update it was determined that the manifest is no longer valid, so the web app has been deleted from the storage and will not be available offline the next time the user attempts to use it.

In a typical situation, we are going to:

- Capture `downloading` so we can show a message to the user and optionally an animated spinner.
- Capture `progress` to make a progress bar.
- Capture `cached` to hide the loading message and tell the user that the app was installed.
- Capture `error` to hide the loading message and tell the user about the situation.
- Capture `updateready` to inform the user that there is an update ready and ask the user if he wants to reload now to access the updated app.



There is no way to polyfill or emulate this API; if a browser does not support application cache it will just ignore the `manifest` attribute and nothing will be downloaded.

We can bind to these events using `addEventListener`. For example:

```
if (window.applicationCache!==undefined) {  
    // The API is available  
  
    applicationCache.addEventListener('updateready', function() {  
  
        // There is an update waiting for reload  
  
        if (confirm("There is an update ready. Do you want to load  
            it now?")) {  
  
            history.reload();  
        }  
  
    }, false);  
}
```

Compatibility and Limits

Fortunately, application cache compatibility is good in the mobile web, but on some platforms issues do arise that are difficult to debug.

Safari on iOS, the Android browser from 2.1, Google Chrome, Amazon Silk, the BlackBerry browser since 6.0, the Nokia Browser for MeeGo, Opera Mobile, Internet Explorer since 10, and Firefox (Figure 13-1) support application cache. Cloud-based browsers do not support this ability, as they need an Internet connection to parse the website on the cloud.

One of the biggest problems with the specification is the lack of a limit definition. How much space can we allocate for an offline web app? Is there any limit? If so, is it per URL or per domain? Usually, up to 5 MB there are no big issues on any platforms.



Application cache has lots of drawbacks and challenges that developers need to solve. Consequently, a group of well-known companies has created the “Fixing Application Cache” group (<http://www.w3.org/community/fixing-appcache/>) with the aim of defining the future of this API.

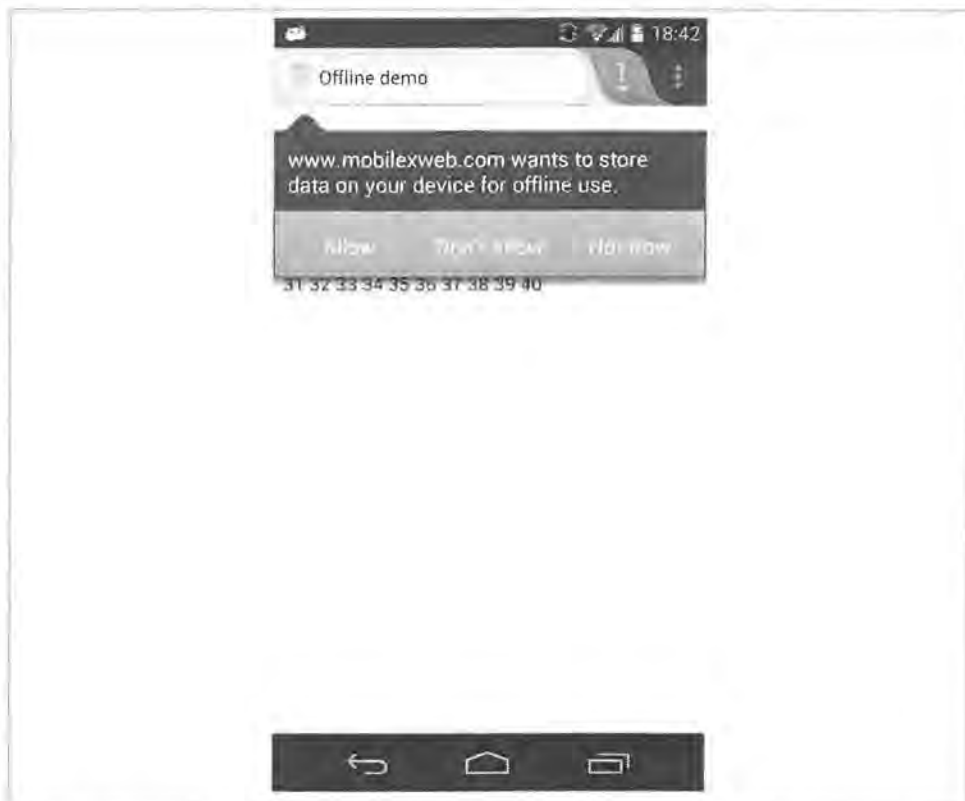


Figure 13-1. Unlike other mobile browsers, Firefox always asks the user's permission before storing an application cache package.

Safari on iOS limits have changed with the different versions, as the following list illustrates:

iOS 3.x

5 MB

iOS 4.x

5 MB without permission, up to 25 MB with the user's permission. In this version, the permission dialog appears when the download exceeds the 5 MB limit; the browser generates an error event even after the user has granted permission, and only on the next reload will the package be stored.

iOS 5.x

5 MB without permission, up to 50 MB with the user's permission. From this version, the browser downloads all the files, asks the user for permission if necessary, and, if permission is granted, save the files persistently without any reload needed.

iOS 6.x

25 MB without permission, up to 50 MB with the user's permission (as seen in Figure 13-2).



Figure 13-2. Safari on iOS will ask for permission to store from 25 MB to 50 MB, since version 6.0.



AppCache limits are per manifest URI; different HTML pages pointing to the same manifest URI will share resources.

From iOS 5, if you provide files larger than the maximum limit (50 MB on iOS 6), the browser will download every file until the limit is reached and will generate an error event after that.

Google Chrome, the Android browser, and Firefox have no predefined limits; the limit is determined by the available space on the device and the amount of time that the user

is willing to wait with the page active. In the case of Opera Mobile, the initial limit is 50 MB, and it can be extended with the user's permission.



Be careful: on some platforms (such as Android), when the device goes to sleep or the user minimizes the browser, window, or tab, the download process will be paused. If you want to store large packages, remember that there are no background download processes; the user needs to maintain the HTML page as the active page while the whole package is being downloaded. If not, the download process will be stopped and the whole package will be invalidated. Therefore, be sure to capture application cache events and warn the user to wait until the package is fully installed (signaled by the cached event).

Only Safari on iOS will continue the manifest package download while in sleep mode or when the user changes tabs or windows.

Client-Side Storage

Working offline is great, but there's a problem: where should a web application store vital statistics and other information when the device is not connected to the Internet? And if the device is not connected to the Internet, how can our applications access helpful databases or information? Client storage solutions come to our assistance, in two flavors: key/value storage and SQL databases (yes, from JavaScript, without server interaction).



Remember that client-side storage mechanisms should be used as temporary and volatile storages, as devices can be restored, lost, stolen, or upgraded. The data should also be stored somewhere else, usually on the cloud (a web server).

Of course, we also have cookies, but they are simpler (only string storage), and we know they are not guaranteed to survive in the browser. However, cookies are the only storage that is shared between the client and the server, through HTTP headers.

Web Storage

The HTML5 Web Storage API defines two key/value stores through two objects: `localStorage` and `sessionStorage`. They are pretty much the same, but the scopes are different: while the local store is used for long-term storage, the session store doesn't persist after the user closes the tab or window.



We should use try/catch blocks when saving items, in case problems occur with the storage or the maximum available space is exceeded.

Both stores are used in the same way:

```
// Save an object or variable in the store
localStorage.setItem("name_in_the_storage", object_to_store);

// Read an object from the store
var object = localStorage.getItem("name_in_the_storage");
```

We can also delete all the objects using `clear` or delete one key using `removeItem`. There is also a storage event that we can listen for with `window.addEventListener` that will be fired when the contents of the local or session stores have changed.

Usually, on mobile devices `localStorage` is the preferred one as the browser session concept is more ambiguous than on desktops. Remember that from a user's perspective the browser may be opened and minimized, but in practice its session may be closed and out of memory. Therefore, `sessionStorage` is useful only for caching objects in memory that are volatile.



If the user is browsing in private or incognito mode, `localStorage` will not persist values. The only exception is Google Chrome on iOS because of a `UIWebView` limitation.

Storing information on the device is a synchronous file operation and it takes time, so it will harm performance if you store small items in a loop. Storing a big item once is preferred in terms of performance in some browsers. Just to give an example, it takes Google Chrome four times longer to store 100 elements of 10 KB each than 1 element of 1,000 KB. Version 2.x of the Android browser can take up to 4 seconds to store 1 MB in web storage.

We can only store string values in the local and session stores. However, as we've seen before, we can convert all of the following to string form:

- Any basic JavaScript value, including numbers and Booleans
- Any noncyclic objects or arrays, converting them to JSON
- Images, converting them to data URI format
- JavaScript code to execute using `eval()`
- CSS styles to be applied in a `style` element
- HTML parsed code to be injected in the DOM

Limits

The HTML5 standard has only an arbitrary recommendation of 5 MB per origin (domains and subdomains) that browsers should implement. However, and this is something I've heard from a browser developer firsthand, "All the browsers are just looking at what the others are doing"—and the de facto standard is to maintain a limit of 5 MB per domain to `localStorage`.

Only Opera Mobile with Presto engine asks for permission to exceed the limit per domain (as shown in Figure 13-3). For other browsers, the user will never receive a confirmation or alert dialog regarding the Web Storage API.

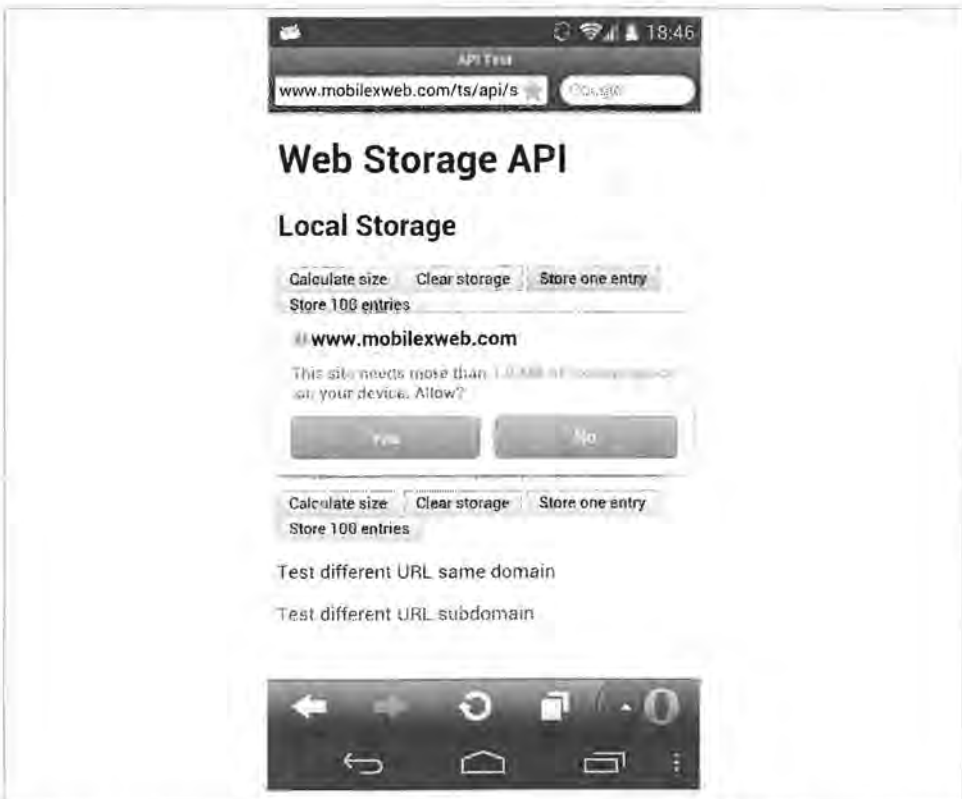


Figure 13-3. Opera Mobile asks for permission to store more than 1 MB in a sequence of powers of 2 (4 MB, 8 MB, 16 MB, 32 MB, etc.).

WebKit-based browsers, such as Safari on iOS, Google Chrome, and the BlackBerry browser, support 5 MB per origin (domain or subdomain), per storage (local or session). However, the strings in WebKit are stored in UTF-16 encoding, so each character takes

up 2 bytes; this gives us a total of 2.5 MB of UTF-8 text. The Android browser has no limits for session storage, but the WebKit limits apply to local storage.



The team working on *The Financial Times*' HTML5 web app has released code to compress and decompress strings to take full advantage of the UTF-16 5 MB storage. Using the simple algorithms (<http://mobilexweb.com/go/ftcompress>), we can store up to 4 MB instead of the 2.5 MB initial limit on WebKit-based browsers.

Table 13-1 shows current limits on the Web Storage API.

Table 13-3. Web Storage API limits per platform

Browser/platform	Initial local storage limit	Session storage limit	Limit can be exceeded
Safari on iOS	2.5 MB UTF-16	2.5 MB UTF-16	No
Android browser	2.5 MB UTF-16	No limit	No
Chrome for Android	2.5 MB UTF-16	2.5 MB UTF-16	No
Nokia Browser for Symbian	2.5 MB UTF-16	2.5 MB UTF-16	No
Nokia Browser for MeeGo	2.5 MB UTF-16	No limit	No
BlackBerry browser for smartphones	2.5 MB UTF-16	No limit	No
BlackBerry browser for PlayBook and BB10	12 MB	12 MB	No
Internet Explorer	5 MB	5 MB	No
Opera Mobile	1 MB	Shared with local	Yes, with user's permission (Figure 13-3)
Firefox	2 MB	No limits	No



Local storage is implemented in the Apache Cordova/PhoneGap library; it replaces the default web view support on some platforms, offering a native storage mechanism.

The Web SQL Database API

The idea of having a relational database available in JavaScript sounds powerful. The WHATWG team has defined an SQL-based asynchronous API called Web SQL Database that is implemented using the open source SQLite engine. When this API came to the W3C discussion table (<http://www.w3.org/TR/webdatabase>), it was abandoned because it doesn't have any other implementation than SQLite, so it can't be a standard. Therefore, more W3C-friendly vendors, such as Mozilla and Microsoft, have decided to not implement this API and instead to go for the standard alternative, IndexedDB.

While IndexedDB is getting more support as the database standard, there are still big opportunities for Web SQL Database in mobile environments, as Safari on iOS up to version 6.0 does not support IndexedDB. The Web SQL API is implemented in Safari, Chrome, the Android browser, the BlackBerry browser, and Opera Mobile; it's not available in Firefox or Internet Explorer.

Even though IE does not support this API, Apache Cordova/PhoneGap supports it in Windows Phone native web app projects, emulating the feature using an SQL Server engine.

The main method that defines the availability of the SQL database is the `window.openDatabase()` method, which creates a new database if the *internalName* doesn't exist, or opens it if it does. This method has the following signature:

```
var db = window.openDatabase(internalName, version, displayName,  
                             sizeExpectable, [callback]);
```

If we use a try/catch, we can capture errors during the operation.



Any script in the same domain can open and edit the same database if it opens it with same *internalName* parameter. The *displayName* is used in some browsers' user interfaces to present a friendlier version of the database name to the user. There is no way to access cross-domain databases.

To execute nonrecordset sentences (CREATE TABLE, INSERT) we can use a transaction using the `transact()` method, which receives a function as a parameter. As a transaction, if one sentence fails, the others will not execute:

```
db.transact(function(t) {  
    t.executeSql('CREATE TABLE countries (id INTEGER NOT NULL PRIMARY  
        KEY AUTOINCREMENT, name TEXT NOT NULL)', [], function() {},  
        errorHandler);  
});
```

The parameter after the query string is an array of parameters to be replaced in the query (using ? inside), the next parameter is the data handler function (not used in a nonrecordset query), and the last parameter is a function handler for errors in the query.



There is no way in this API to provide an already created database; all the databases and tables must be created from code and we can't choose where to store them.

To create a typical SELECT statement with recordset looping, we can use the following template:

```

db.transact(function(t) {
    t.executeSql('SELECT * FROM countries', [], countriesHandler,
        errorHandler);
});

function countriesHandler(transaction, data) {
    var record;
    var id;
    var name;
    for (var i=0; i<data.rows.length; i++) {
        // We get the current record
        record = data.rows[i];
        id = record['id'];
        name = record['name'];
        // Do something with record information
    }
}

function errorHandler(transaction, error) {
    alert('Error getting results');
}

```



The Google Gears project, which predates HTML5, tried to provide similar behaviors, including offline and SQL database support. Gears was compatible with Android 1.x and some BlackBerry devices; it was abandoned by Google in the wake of HTML5.

If you allow working offline you should implement a synchronization method, using Ajax to download changes from and upload them to the server.

Limits

Apple was the pioneer of the implementation of this API on the iPhone, and today it's still the platform that allows us to have the most control over limits. Safari on iOS supports up to 5 MB per database without requiring the user's permission, and we can extend that limit to 10, 25, or 50 MB if the user accepts it. Other compatible browsers have no specific limits, and usually up to 50 MB there are no problems.

As SQLite uses a binary file, we don't have the problem we had with local storage in terms of UTF-16 usage, and we can store not only strings, but also numbers and binary data.



One of the key problems of the SQL Database API is that there is no way to enumerate or delete a database once it's been created. We can always drop all the tables, but if we've reserved some space, the browser should still save it for the database. The user is in charge of accessing the browser's settings and removing the database.

We define how much space we want to reserve using the fourth parameter of the `openDatabase()` method. Therefore, if we want to receive permission for a 45 MB database ($45 \times 1,024 \times 1,024$ bytes), we can try to open (or create) it and the user will be asked for permission, as we can see in Figure 13-4:

```
var db = window.openDatabase("mydata", "1.0", "My data", 45*1024*1024);
```



Figure 13-4. With Safari on iOS we can ask for the user's permission to store large amounts of data (from 5 to 50 MB).

If we increase the database size automatically, the user may approve the new limit when the current threshold is surpassed, but it's a better idea to pre-reserve all the space we want.



Lawnchair (<http://brian.io/lawnchair/>) is a local client JSON store that uses HTML5 features behind a unique and very simple API. It uses local storage, IndexedDB, Web SQL, the FileSystem API, or other hacks on older platforms.

The IndexedDB API

The IndexedDB (<http://www.w3.org/TR/IndexedDB>) or IDB API is the W3C standard replacement for the Web SQL Database API. IDB is not yet available in Safari on iOS before version 6.0 or the Android browser prior to version 4.1, but it's implemented in Chrome, Firefox, and Internet Explorer (from version 10), and with time it will probably replace Web SQL on all platforms.

The IDB API allows us to create and query databases, asynchronously or synchronously (useful for web workers). Unfortunately, at first sight the API doesn't look so nice or easy to understand. The interface is a NoSQL (unstructured) database, which can take some getting used to if you're used to working with relational databases.

The basic concepts include a *database repository* having multiple *object stores* (similar to tables) that contain *objects* (similar to records) indexed with a *key* (similar to a primary key). A database, or store, can only be accessed by a same-origin policy, as in the other storage APIs, meaning that we can't access databases cross-domain. IndexedDB provides support for objects to manage indexes, tables, and cursors in a particular transaction. The transaction is the only place where we can execute commands. With IDB there is no way to execute SQL commands.



Using Modernizr we can query about storage API support using the properties `localStorage`, `sessionStorage`, `webkitDatabase`, `indexeddb`, and `applicationcache`.

While covering the full functionality of IDB is outside the scope of this book, we'll take a quick look at the basics so you can see how to create a database, create an object store, and work with data.

The main functionality happens in the global `indexedDB` object. Some versions of Firefox, the BlackBerry 10 browser, and Chrome up to version 18 use prefixed versions, so to be compatible with previous versions we should check for the prefix first or reassign the global object, as in:

```
window.indexedDB = window.indexedDB || window.webkitIndexedDB || window.mozIndexedDB;
```

Opening a database is done using `indexedDB.open()` and we need to bind events using `on<x>` syntax or `addEventListener`, as in:

```

var dbRequest = indexedDB.open("myDatabase");
var database;
dbRequest.onsuccess = function(event) {
    // The database object appears as event.result
    database = event.result;
}
dbRequest.onerror = function(event) {
    // Manage error
}

```

A big difference between Web SQL and IDB is that when we call the `open()` method in IDB the database object is not returned. It is available as the `result` property of the success event parameter.

To create the object stores and schemas we need to implement the `upgradeneeded` event, which will also be fired when upgrading database versions:

```

dbRequest.onupgradeneeded = function(event) {
    var db = event.target.result;
    // Let's create the object store for books
    var objectStore = db.createObjectStore("books", { keyPath:
        "isbn" });
}

```

To execute commands on an object store we need to use transactions, which receive as parameters a single store name, an array of store names, or an empty array (for all stores) to query on, and the operation type (by default, read-only), as in:

```

var transaction = database.transaction("books", 'readwrite');
transaction.objectStore("books")
    .add({ isbn: "1234567890", name: "Programming the Mobile
        Web"});

```

To get one element by key from the database, we can use a chained syntax, as in:

```

database.transaction("books")
    .objectStore("books")
    .get("1234567890")
    .onsuccess = function(event) {
        var object = event.target.result;
    });

```

To browse between all objects, we can use a cursor:

```

var allBooks = database.transaction("books").objectStore("books");

allBooks.openCursor().onsuccess = function(event) {
    var cursor = event.target.results;
    if (cursor) {
        var key = cursor.key;
        var currentBook = cursor.value;
        cursor.continue(); // Move to next element and execute
                           // onsuccess again
    } else {

```

```

    }
    // End of object store
};

```



Internet Explorer 9 has no support for Web SQL or IndexedDB. Therefore, the only persistent storage is local storage.

The FileSystem API

The FileSystem API (<http://www.w3.org/TR/file-system-api>), also known as “File API: Directories and System,” is a W3C draft specification—more experimental in terms of storage compatibility—that allows our domain to have its own full virtual filesystem in the device memory. We must not confuse this API with the File API (useful to read files from the user’s filesystem), which we will cover in Chapter 15.

With this API we can create and manage folders and files in a virtual filesystem that may or may not have a corresponding private place in the real device filesystem. Some browsers may implement it using a database. The API does not allow us to work with the user’s filesystem (private or public); we can only work with the files created by scripts in our domain.

We can request two kinds of filesystem, PERSISTENT or TEMPORARY, and create our own structure inside.

To retrieve a persistent filesystem of 4 MB, we can use the following code:

```

window.requestFileSystem(window.PERSISTENT, 4*1024*1024,
function(filesystem) {
    // The filesystem was created/opened

}, function(event) {
    // Handle error
});

```

With this API we can:

- Modify filesystem quotas
- Create and open files
- Append, replace, and delete files
- Create and delete directories
- Copy, rename, and move files and directories
- Read directories’ content
- Create filesystem URLs

With this API we also have a new URI, *filesystem:*, that we can use to point from our web app to any file stored in our persistent or temporary storage.

At the time of this writing, only Google Chrome for Android supports this API; check Mobile HTML5 (<http://mobilehtml5.org>) to get updated compatibility information.



In Chapter 15, we will cover the File API, which can be used in many situations, including with the FileSystem API's files.

User Intervention

While it may seem that we are in charge of all the storage APIs, the truth is that most modern mobile web browsers allow the user to see and delete a website's data. While the full details are not always given, at least the domain and how much space all the storage APIs are consuming is visible.

In Figure 13-5, we can see Safari's Advanced→Website Data and Google Chrome's Content→“Website settings” screens.



Figure 13-5. Some mobile browsers allow the user to see and delete website data, including AppCache and storage data.

Debugging Storage

Debugging storage (application cache, local storage, databases) can be a little tricky sometimes. We will cover debugging tools in Chapter 18, but for now it's important to

know that remote inspection tools, such as the Safari, Chrome, and BlackBerry developer tools, allow us to see all the storage's content and even execute SQL commands on databases, as seen in Figure 13-6.

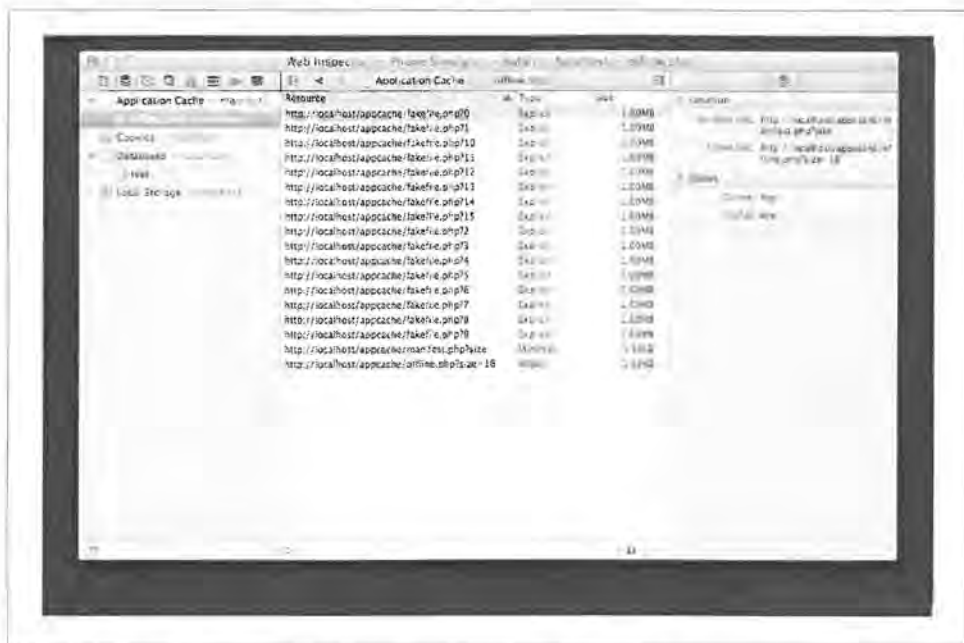


Figure 13-6. With remote debugging sessions we can check the current storage status for every API covered in this book.

Network Communication

Communication between our mobile web apps and a server is usually a must-have feature. Modern mobile browsers have different APIs that will help us with this task, including the well known Ajax and HTML5 APIs.

Ajax

Ajax, originally an acronym of *Asynchronous JavaScript and XML*, is a technique that involves making asynchronous server requests without refreshing the page, interrupting the user's activity, changing the browser's history, or losing global state variables.



Why did I say “originally an acronym”? Today the term *Ajax* is used in a more general way to define interactive Web 2.0 applications that use asynchronous requests to the server, but may or may not be written in XML. I even hear a lot about Ajax in dynamic websites using the jQuery UI, ExtJS, or other rich control libraries that don’t actually make background requests to the server.

The magic behind Ajax is called XMLHttpRequest; it is a native JavaScript object available in compatible devices that was based on an ActiveX object created by Microsoft in Internet Explorer 5.0. At the time of this writing, basic Ajax is supported in just about every mobile browser out there. By basic support, I mean XMLHttpRequest 1.0 support with the ability to download text (sometimes treated as JSON) and XML.

XMLHttpRequest Level 2 (XHR 2, sometimes called Ajax 2) is an extension to the original standard defined by the W3C (<http://www.w3.org/TR/XMLHttpRequest>). With this extension, we can manage a new load event instead of managing states in readystate change; it also supports the progress event as well as the ability to upload and download binary information.

Using XHR 2, a normal Ajax request looks like:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'myurl', true);
xhr.onload = function(event) {
    // xhr.responseText includes the server's response as Text
}
xhr.send();
```

XHR 2 also added the `responseType` attribute, allowing us to read the result in different formats. We can set this to `arraybuffer`, `blob`, `document`, or `json`.



Using a Modernizr extension we can detect XMLHttpRequest Level 2 and CORS support using the `xhr2` and `cors` properties. We can also verify current compatibility at www.mobilehtml5.org.

Cross-domain requests

While not part of XHR directly, CORS (Cross Origin Resource Sharing) is a way to make Ajax compatible with cross-domain servers. By default, XHR allows calls only to the server on which the resource making the requests resides. With the CORS specification (<http://www.w3.org/TR/cors/>), a cross-domain server can define a couple of headers in the response to allow our or any website to make requests to it using Ajax calls.



If your mobile website gets content using Ajax, you should implement Google's proposal for search engine optimization (SEO). You can find more information about this at code.google.com/web/ajaxcrawling.

For example, to allow a script to accept requests from any website, we should deliver the following HTTP header:

```
Access-Control-Allow-Origin: *
```

This is particularly useful to use XHR to call third-party APIs, such as Twitter, Flickr, or Google Maps.

JSONP

JSON with Padding (JSONP) is a hack for accessing a third-party domain's content without the cross-domain problems of Ajax requests to servers on which CORS is not enabled. Many public web services are offering this new way of communicating with third-party servers.

JSONP uses a script tag generated by JavaScript pointing to a URL with a parameter we define, generally for a local callback function to be called when the script (and the data it fetches) is downloaded and executed.

A very similar technique is used for scripting code: you download only a subset of the scripts in the initial download, and then you download the other scripts that you will need later.

JSONP needs one feature to be working on the browser: the ability to insert a script dynamically from JavaScript. If this feature works, the browser should detect the new DOM script element and automatically download and execute this new resource. As this script will call your function with the data, you will be able to receive data from a third-party server.



Modern JavaScript libraries such as jQuery support JSONP requests without dealing with the DOM. You can use `$.getJSON` with a parameter to replace an Ajax query with a JSONP query.

Generally, the third-party server offers some URL to use JSONP as a format. For example:

```
http://api.thirdpartyserver.domain/<jsonp_script>?<callback>=  
<our_function>
```

The server will respond with something like this:

```
<our_function>({<json_data>});
```

We can test if a mobile browser detects the dynamic creation of script elements using the following code:

```
function doJSONP() {  
    var head = document.getElementsByTagName('head')[0];  
    var script = document.createElement('script');  
    script.type = 'text/javascript';  
    script.src = 'http://mobilexweb.com/tests/jsonp?cb=finished';  
    head.appendChild(script);  
}
```

Ajax with offline resources

Ajax can be used when we are in offline mode in two situations:

- When using web views, such as in Apache Cordova/PhoneGap apps. We can load resources from our application bundle (JSON, text, XML, or binary information) using Ajax, even if there is no web server running.
- When using application cache, if the URL we are trying to get is listed in the manifest file.

Server Sent Events

Server Sent Events (<http://dev.w3.org/html5/eventsource>) (SSE, also known as *EventSource*) is a network specification draft for enabling web apps to use HTTP for uni-directional real-time communication from the server to the browser. Basically, it's like an Ajax request that doesn't close the connection, so the server can send pieces of information to the client whenever it needs to.

This API aims to replace the long-polling Ajax techniques, sometimes known as Comet techniques, which are not standard and may not work properly on mobile devices.

If for some reason the client or the server closes the connection, the browser starts a polling mechanism, opening the connection again.

At the time of this writing, SSE is compatible with Safari on iOS (since 4.1), Google Chrome, Amazon Silk 2.0, the BlackBerry browser for PlayBook 2.0 and BB10, Nokia Browser for MeeGo, Opera Mobile, and Firefox.

To create an SSE request, we must instantiate an `EventSource` object with the script URL and using a messaging API event mechanism, as follows:

```
var request = new EventSource("/mySSEscript");  
request.onopen = function() {  
    // An HTTP request has been opened. It can be called more than  
    // once if the connection drops  
}  
request.onerror = function() {
```

```

    // Something went wrong or the connection has been dropped (and
    // it will try to reconnect)
}
request.onmessage = function(event) {
    // The server has just sent a message through the connection and
    // it's available in the data property
    var data = event.data;
}

// If we want to close the connection later, we call request.close()

```

Every time the server sends a message, we will receive a message event handler call. The message data can only be a string.

The server component

Usually this kind of real-time communication mechanism requires a server-side platform prepared for this kind of work (clients connected for long periods of time), so web servers like Apache or IIS may not be well suited to it. Solutions such as Node.js are much more optimized and prepared. However, SSE can be implemented with any server-side HTTP platform—even PHP, as it's mounted over HTTP.

To answer an SSE request, we must use the MIME type `text/event-stream` and follow the SSE protocol. The basic protocol involves answering with `data: <data>` and a line break. We replace `<data>` with the message we want to send to the client, which will be received in the message event. After sending a message, instead of closing the connection, the server should keep waiting in an event loop for new messages to send. Therefore, the server and the client are permanently connected, like a server that is taking a long time to respond and periodically flushing small pieces of information.



Besides the `data:` line, we can also define an `id` that will be used by the client when the connection drops to recover the same session.

In PHP, we can do something like:

```

<?php
header('Cache-Control: no-cache');
header('Content-Type: text/event-stream');
$send = false;

while (!$send) {
    // Wait for some external event here

    echo 'data: Hello from the server';
    echo '\n\n'; // New line
}

```

```
flush();  
?>
```

We are using an `$end` Boolean variable here that we need to set to true when we want to end the connection. In the meantime, we should wait for something to happen. If you just want a heartbeat, you can use the `sleep()` PHP function to send messages to the client every *x* seconds. In our example, the `message` event will receive the `Hello` from the server data string.

WebSockets

While SSE is great for real-time apps, it has a big problem: the communication can't be done from the client to the server. WebSockets solve this problem, enabling bidirectional, full-duplex, real-time communication between the browser and a web server.

WebSockets involve two specifications: the WebSocket API (<http://www.w3.org/TR/websockets>), which defines the JavaScript API, and the WebSocket network protocol (<http://tools.ietf.org/html/rfc6455>). Because of their bidirectional nature, WebSockets do not use standard HTTP messages, and therefore a special server is required.



There are some Apache-based projects to add WebSocket protocol supports, such as Apache WebSocket (<https://github.com/disconnect/apache-websocket>); on the Microsoft side, this protocol was added in Internet Information Server (IIS) version 8 in conjunction with ASP.NET 4.5.

One of the biggest problems of this specification today is that there were many protocol versions during the draft stage, before it was finished by the IETF organization under RFC 6455. Consequently, different browsers may implement different versions. Previous versions of the protocol are known as *hixie-75*, *hixie-76/hybi-00*, *hybi-07*, and *hybi-10*.

At the time of this writing, the WebSocket API is available in Safari on iOS (since 4.2), Google Chrome for Android, Amazon Silk (from 2.0), the BlackBerry browser (from 6.1, BB10 and PlayBook included), Internet Explorer (since 10), Opera Mobile (since 12.1), webOS on the HP TouchPad, and Firefox.



We define WebSocket URLs using the `ws://` protocol definition.

From an API perspective, using WebSockets is easy: we just instantiate a `WebSocket` object and use the `message` event and the `send()` function to receive and send

information from/to the server. When we create a WebSocket connection we can optionally define the protocol name as the second argument:

```
var socket = new WebSocket("ws://myserver.com");
socket.onopen = function() {
    // The connection was opened
}
socket.onclose = function() {
    // The connection was closed
}
socket.onmessage = function(event) {
    // We have just received a message from the server
    var message = event.data;
}
// We may send messages
socket.send("Hello server!");
// We can close the connection from the client
socket.close();
```



The WebSocket API is not like low-level socket implementations in other languages, such as C or Java. It supports only string messages, although there is a nonstandard extension supporting binary JavaScript data that works in some browsers.

We need to provide a server solution for our WebSockets. There are frameworks available for almost any server-side language and platform, but remember that WebSockets are more suitable for event-driven platforms that can maintain several connections at the same time instead of closing the connection after the response.

Socket.IO

Socket.IO (<http://socket.io>) is a powerful open source client- and server-side framework, based on Node.js. Created by Guillermo Rauch (<http://www.devthought.com>), it offers socket abilities using different techniques transparently to the client and server developer, through a very simple but unique API. Socket.IO uses WebSockets but will fall back to other mechanisms based on current browser compatibility, including an invisible Flash socket engine, Ajax, multipart Ajax, forever iframes, or JSONP polling techniques.



Pusher (<http://pusher.com>) is a cloud-based service that allows WebSocket servers to scale easily. It offers a free service supporting up to 20 concurrent connections and 100,000 messages per day, and it also offers commercial services based on the project's needs. It offers a client API and a server API, both communicating with the Pusher infrastructure.

Geolocation and Maps

One of the great features of mobile devices is that they can go everywhere with us. That is why the *where* is a very important context to be considered by our websites. Knowing the user's location can help us to show useful contextual information. If I live in London, why should I receive a banner promotion from a shop in New York? Likewise, if I am on holiday in Singapore and I search for "pizza," I would like to receive relevant information about where I can get it.

Location-based services (LBS) are one of the key features of modern mobile web applications. From our mobile websites, we can get the user's location using many techniques. Mapping and LBS are very popular right now, so it is easy to find web services and APIs from different providers that we can integrate into our mobile websites.

Location Techniques

There are different techniques that we can use to determine the geographical location of a device, based on the platform, the browser, the operator, and so on. Most technologies involve server detection, but others depend on client detection, and we may even rely on the user's input.

Accuracy

Every location technology has some accuracy error. This is usually specified in a distance metric, like meters or kilometers, but in some techniques accuracy is defined according to levels, such as *city accuracy* or *country accuracy*.

Indoor Location

When we think about geolocation, we tend to think about the outside location—that is, where on the planet the user is located. Recently, there has been some interest in services

that will locate a user inside a building (for example, a shopping mall or an office building). The idea is that we can offer better services for the users inside the building if we can pinpoint what floor they are on or what department they are in. These services may be local services on the Internet, or even services provided on an intranet with the user using a Wireless LAN connection.

Augmented Reality, the Promise

Augmented Reality (AR) is a very popular technology today. It involves the usage of the camera preview with additional information on the screen about the objects and places we are seeing. One of the AR solutions is geographical-based, allowing us to see, for example, the camera preview with a tag over every building and point of interest (POI), with more information displayed about those places.

This solution involves an accelerometer, high-accuracy geolocation (such as GPS), a digital compass, and camera support. Unfortunately, web applications cannot access all of these features on a mobile device today, so we cannot create AR web apps yet; however, it will be possible in the near future.

Client Techniques

Devices support a wide variety of approaches to figuring out where they are. Different approaches may yield different results.

GPS

The Global Positioning System (GPS) is the first technique most people think of when location detection is mentioned. The United States government created GPS as a system for locating devices, using between 24 and 32 satellites orbiting the Earth. Many mobile devices come with a built-in GPS receiver that can read satellite data to determine location information (data must be received from a minimum of four satellites). In mobile devices, the accuracy error is between 2 m and 100 m. The user needs to have a sky view (outside), and it can take between 5 seconds and 5 minutes to calculate the location.

A-GPS

Assisted GPS (A-GPS) is a software-based system available for mobile phones connected to carrier networks that can help the devices to determine their locations. The assistance can be in the form of helping the device to find a better satellite signal, or providing less-accurate information about the location of the user until the GPS has connected successfully.

In 2006, I started to use a Nokia N95 with GPS support. In my city, it took 5 minutes to get my location using GPS (with an accuracy error of 10 meters). A firmware update

later added A-GPS support to the same hardware, allowing the same device in the same city to connect in 10 seconds, with an initial accuracy error of 100 meters.

Cell information

Using the operator network's cellular towers, the carrier can triangulate the position of a mobile device. The accuracy will depend on how many cell towers are in range (the more densely populated your location is, the more towers will be in range and the more accurate the reading will be). The carrier knows every cell tower's position, so it can make the required calculations to detect the device's location.

Even knowing which cell tower a device is connected to can provide an idea of its location (near the location of the tower). This might be accurate to within a block, or up to some kilometers in rural areas.

Getting the Cell Location Without the Carrier's Assistance

It is possible to detect a mobile device's position using cell (tower) information without the cooperation of the carrier. OpenCellID (<http://www.opencellid.org>) is an open source project aiming to create a complete database of cell IDs worldwide. If we can get the IDs of every cell in range and calculate the distances to those cells, we can triangulate the device's position.

For example, Google Maps can locate even non-GPS-equipped devices anywhere in the world, thanks to some carrier agreements for cell detection.

WiFi Positioning System

If you have a notebook with WiFi, go to Google Maps (<http://maps.google.com>) and click on the blue circle (the locate me feature). If you are in a large city, you will probably be located very accurately. You were just geolocated, and unless you have a 3G netbook, chances are your notebook doesn't have GPS. This technique also works on a WiFi-connected iPod touch and on tablets. But how?

The WiFi Positioning System (WPS) is a very clever technique that detects your location using the list of wireless routers that are available in your area (even if you are not connected to them). This method relies on a pre-existing database of routers and their geographical locations. Skyhook Wireless (<http://skyhookwireless.com>) is the leading provider, offering developer programs for most mobile and desktop platforms. Google has its own database and is also the provider used by Firefox.

The main problem for us is that as yet there are no mobile browsers that give us the hotspot list.

Server Techniques

On the server, we can get the HTTP request headers. This is our opportunity to locate the user without using any client technology, such as GPS, and in a way that works even for low-end devices.

IP address

The main server technique for locating a user is reading the client's IP address. However, this is not as straightforward as it may sound. Depending on the user's connection type (2G, 3G, WiFi), the IP address we receive may be the operator's WAP gateway address, a dynamic IP address in the operator's range, or the IP address of the WiFi connection.

To further complicate our work, we need to bear in mind proxied browsers (discussed in Chapter 2). These browsers use a proxy server to connect to the Internet and to our servers. For example, if the user is browsing using Opera Mini, we will receive the requests from the Opera server instead of from the user's device. Likewise, if the user has a BlackBerry device and is using a corporate Internet connection, we will receive the requests from that connection, which could be based thousands of miles from the user's actual location.

What should we do with the IP address? There are public lists of operators' IP addresses, and there are public and commercial solutions for determining the location of an IP address. The accuracy of this method can be country-level to city-level, although in some special situations, like when the user is using a public WiFi network, we can pinpoint the exact location.

Carrier connection

Some worldwide operators offer developer programs (both open and private) for web portals that allow any request made from a user to your web server to carry additional headers containing information about the user (identity, location, and billing services). The GSM Association, which encompasses almost all the operators around the world, has launched an initiative called *OneAPI* that aims to provide web applications with access to all this carrier information through its APIs.

Language

A less accurate mechanism is to use the accepted language of the browser. If the user has set up his device correctly, it should send a header indicating the preferred language, from which we can infer what country the user is in (for example, the browser may send us en-CA as the accepted language, meaning English from Canada). This results in at best country-level accuracy.

Indoor location

When users are connecting via WiFi hotspots in a single building, we can configure our routers to be queried about those users. Every WLAN user has a unique IP address in the network, so we can tell which hotspot a given user is connected to. With that information, we can identify the floor and zone where the user is located.

Asking the User

If you need to offer web-based location-based services, the last location mechanism available if all the others fail is to *ask the user*. Even if you have identified the approximate location using another algorithm, you may be able to increase the accuracy by asking the user.



The user may know her location, or not. For example, if the user is visiting a foreign city, she may not know her exact current location.

So, what should we be asking users? We should allow them to select from a closed list, or to type the location into an open text box. We can query our databases for city names, addresses, POIs (such as parks, hotels, or restaurants), or zip codes. We can also query public databases using web APIs like Yahoo! PlaceMaker, FourSquare, and the Google Places API. To pinpoint the location, we can then use a geocoding query to convert the string into a latitude/longitude pair.

We should allow the users to select their current location from amongst the following:

Home

If the users need to log in to the website, we can ask them where they live when they install the application and store this information in our database for future use.

Favorite places

We can make a users' favorite places database.

History

We can allow the users to select places where they have been recently, ordering the list by frequency.

Records of all of these locations may be stored in a database on our server attached to the user's credentials (for login-based solutions), in a cookie on the client, or even in client storage on devices that support this.

Detecting the Location

We have two options for detecting the user's geographical location: using the W3C Geolocation API, available under the HTML5 APIs, or using third-party APIs, usually available directly from carriers.

The W3C Geolocation API

The W3C standard way to query the user's position from JavaScript is called the Geolocation API (<http://www.w3.org/TR/geolocation-API>). The Geolocation API doesn't rely on one location technology. Instead, it allows the browser to decide which method it will use.

With this API implemented in a mobile browser, the `navigator` object in JavaScript will have a read-only property called `geolocation` that will allow us to interact with the API.

Location querying is an asynchronous process. It can take some time to get the user's location (like with GPS); that's why the API relies on callback functions to give us the latitude and longitude.

The user will need to give the site permission to obtain the geolocation data using the API, as shown in Figure 14-1.

Getting the position

The first way to use the Geolocation API is to get the user's location using the `getCurrentPosition()` function of the `geolocation` object. It receives two callbacks: the function that will receive the position, and an error-handling function. The latter is optional.

Optionally, it may also receive an object that configures some additional properties; this third parameter will be discussed shortly.

Let's look at an example:

```
navigator.geolocation.getCurrentPosition(userLocated, locationError);
```

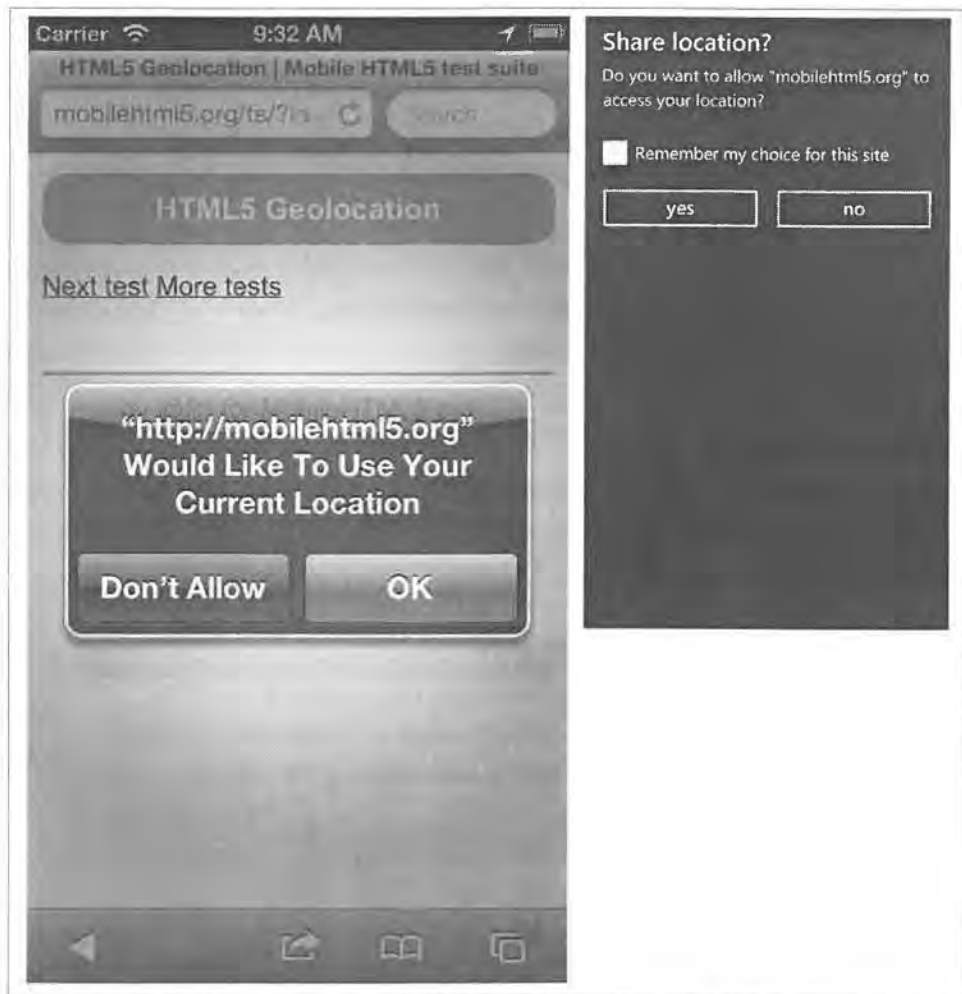


Figure 14-1. We cannot get the user's location unless permission was granted the first time we tried to get it.

The first callback will receive one parameter as the position object with a coordinate property. The error callback will receive an error code:

```
function userLocated(position) {
    var latitude = position.coords.latitude;
    var longitude = position.coords.longitude;
    var timeOfLocation = position.timestamp;
}

function locationError(error) {
    alert(error.code);
}
```

The `coords` property has the following attributes, defined in the W3C standard:

- `latitude` in decimal degrees
- `longitude` in decimal degrees
- `altitude` (optional) in meters above the ellipsoid
- `accuracy` in meters
- `altitudeAccuracy` (optional) in meters
- `heading` (optional) in degrees clockwise related to true north
- `speed` (optional) in meters per second

Handling error messages

The parameter received in the error handler is an object of class `PositionError` having a `code` and a `message` (useful for logging). The class also has some constant values to be compared with the `code` property. The constants are shown in Table 14-1.

Table 14-1. *PositionError constants in the W3C Geolocation API*

Error constant	Description
<code>PERMISSION_DENIED</code>	The user has denied permission to the API to get the position.
<code>POSITION_UNAVAILABLE</code>	The user's position couldn't be determined due to a failure in the location provider.
<code>TIMEOUT</code>	The user's position couldn't be determined before the timeout defined in the options.

To use these constants, we should make a switch for each value:

```
function locationError(error) {  
    switch(error.code) {  
        case error.PERMISSION_DENIED:  
            // error handling  
            break;  
        case error.POSITION_UNAVAILABLE:  
            // error handling  
            break;  
        case error.TIMEOUT:  
            // error handling  
            break;  
    }  
}
```

Tracking the location

The second way to use the W3C Geolocation API is to track the user's location. With tracking support, we can receive notifications about location changes. For instance, we can make a sports website that tracks the user's steps, makes speed and distance calculations, and stores this information either locally or on our server using Ajax.



For this to work, the user needs to keep the website open in the browser. Also be aware that many browsers stop JavaScript execution when the browser is in the background, as discussed in Chapter 12.

The tracking process involves the `watchPosition()` method of the `navigator.geolocation` object, which receives two handlers (for location detection and error management) and returns a `watchId`. The handler function will receive the same parameter as the `getCurrentPosition()` function that we saw earlier. To stop the location tracking we can call `clearWatch()`, passing the previously received `watchId`:

```
// Global variable to store the watch ID
var watchId = false;

// This function may be called by an HTML element
function trackingButtonClick() {
    if (watchId==false) {
        // Tracking is off, turn it on
        var watchId = navigator.geolocation.watchPosition(userLocated,
            locationError);
    } else {
        // Tracking is on, turn it off
        navigator.geolocation.clearWatch(watchId);
        watchId = false;
    }
}
```



Using Modernizr, we can check for geolocation support with the `geoLocation` property; we can also use server-side detection with WURFL, querying whether the `ajax_preferred_geoloc_api` capability has a value of `w3c_api`.

Defining optional attributes

The third parameter of the `getCurrentPosition()` and `watchPosition()` functions can receive an object with the optional properties outlined in Table 14-2.

Table 14-2. *Optional properties for `getCurrentPosition()` and `watchPosition()`*

Property	Type	Default value
<code>enableHighAccuracy</code>	Boolean	false
<code>timeout</code>	Long (in milliseconds)	Infinity
<code>maximumAge</code>	Long (in milliseconds)	0

If the `enableHighAccuracy` property is defined as `true`, the provider should force the best accuracy in determining the user's location (such as GPS if available).



When developing Apache Cordova/PhoneGap applications for Android, you must define `enableHighAccuracy` as `true` if you want to test geolocation in the Android emulator.

The `maximumAge` property is useful when retrieving location data cached on the device. With this property, we can specify how recent we want the cached data to be—if the data is older than the specified maximum age (in milliseconds), the device will have to get a new location. A typical usage might look like this:

```
navigator.geolocation.getCurrentPosition(userLocated, locationError,
    {timeout:10000, maximumAge: 30000, enableHighAccuracy:false});
```

If the `maximumAge` property is defined as 0 (the default value), the device must always acquire a new location when `getCurrentPosition()` is called.



Some older browsers—such as Android 1.x, BlackBerry browser 5.0, and Opera Mobile 9.5—don't support the Geolocation API, but they used to support the Google Gears plug-in, which has a very similar implementation to the W3C standard.

The BlackBerry browser before version 5.0 used the proprietary `blackberry.location` object, which is faster and less battery-intensive but provides less accuracy and information than Gears or the W3C standard.

Geolocation API 2.0

The W3C is working on the next generation of the Geolocation API (<http://dev.w3.org/geo/api/spec-source-v2>), including some additions to the current specification (mainly the ability to query about postal addresses).

There are two optional Boolean attributes that we can use when we query about locations: `requireCoords` and `requestAddress`. The first forces the system to deliver coordinates, and the second defines whether or not we want to receive a civic address (street, number, city, country).

When we get a location through `getCurrentPosition()` or `watchPosition()`, there is an optional address attribute in the argument (in the same place where `coords` exists). The address object may contain a set of properties including `country`, `region`, `county`, `city`, `street`, `streetNumber`, `premises`, and `postalCode`. Every property is optional, so how much information the browser will give us will depend on the current situation.

At the time of this writing, no mobile browser supports these features; however, we can emulate them using a provider API, such as the Google Maps API.

Carrier Network Location APIs

Some carriers offer custom APIs that developers can use for free, for a monthly fee, or for a geolocation request fee.

GSMA OneAPI

OneAPI is a cross-operator API organized by the GSM Association. With this API we can access the user's location from our servers, using his phone number. Register as a developer and obtain a token to access the OpenAPI web services (<http://oneapi.aepo-na.com>). The API supports SOAP Web Services and REST using HTTP.

One of the services supported by OpenAPI is geolocation. With the Location API, we can get a user's longitude and latitude using the mobile operator's cells' positions. To use the API we need to get a key from the website. Then, if we want to use REST, we can create an HTTP request to a URL like the following:

```
https://developer.aepo-na.com/TerminalLocationService/Proxy/REST/<key>?
address=tel:<tel>&accuracy=coarse
```

where *<key>* is the key assigned to our developer account and *<tel>* is the international number of the phone we want to geolocate. If the request is successful, we will receive a response like the following:

```
<response timestamp="2010-06-06T12:31:07.014Z" longitude="10.22244"
latitude="54.601505" altitude="10.0" accuracy="200"/>
```

The list of supported operators is on the website, and the goal is to have all the operators worldwide using the same API.

Specific carriers' APIs

Table 14-3 shows other carrier geolocation APIs that we may use, using the network services after getting the user's permission.

Table 14-3. Carrier geolocation API availability

Platform	Carriers	URL
Verizon Network API	Verizon (US)	developer.verizon.com
BlueVia	Movistar (Latin America, Spain), O2 (UK, Germany), Telenor (Asia, Scandinavia, Eastern Europe)	bluevia.com
AT&T Location API	AT&T (US)	developer.att.com/developer
LBS Sprint API	Sprint (US)	developer.sprint.com
Orange Location API	Orange (France only at the time of this writing)	api.orange.com



LocAid (<http://developer.loc-aid.com>) offers a premium solution that can locate 350 million devices with one API from AT&T, Sprint, T-Mobile, and Verizon. Location Labs (<http://developer.veriplace.com>) offers the ability to locate 250 million devices with a single API too.

IP Geolocation

There are a lot of free and commercial IP address geolocation services available for use on our servers. When using such a solution, we need to remember that a BlackBerry can browse through a corporate network, so the IP address will be the network IP address and not the user's. The same applies to proxied browsers like Opera Mini.

Reading the IP address

We can read the IP address from the host using the appropriate mechanism for the server platform. For example, in PHP we read the address using:

```
$IP = $_SERVER['REMOTE_ADDR'];
```

However, we must remember that this IP address may belong to a renderer proxy. For example, when an Opera Mini user accesses our website the IP address will always be the same, because the client contacting our server is actually the Opera Mini server. Fortunately, Opera Mini servers offer us another HTTP header that provides the actual IP address of the requesting mobile device: the X-Forwarded-For header contains a CSV list of the IP addresses of all the proxy servers the request has passed through on its way from the device to the Mini proxy, and the last IP address will be the address of the original requestor (the mobile device).

Once we have the IP address to query, we can use a web service to get the country/city details, or download the Geo-IP open source database (<http://software77.net/geo-ip>).



We need to keep in mind that IP geocoding is useful only to get the user's country for devices connected to the Internet via 2.5G or 3G, because what we'll receive is the operator's gateway IP address. If the user is using WiFi, depending on the zone, we can usually get more accurate details.

Google's ClientLocation object

Google provides a set of JavaScript APIs (Maps, Search) that can be used to create feature-rich dynamic websites. Whenever one of these APIs is loaded on a client, the Google Loader attempts to geolocate the user using the device's IP address.

To add support for the APIs, we need to insert this script:

```
<script src="https://www.google.com/jsapi"></script>
```

To use the client location feature, we must then load an API. For example:

```
<script type="text/javascript">
  google.load("search", "1");
</script>
```

Once we've loaded the API, the `google.loader.ClientLocation` object will be populated with properties like the following:

- `latitude`
- `longitude`
- `address.city`
- `address.country`
- `address.country_code`
- `address.region`



This technique works only on compatible devices and should be used only if we are going to make use of one of the Ajax APIs (Ajax Search, Maps, Ajax Feeds, Earth, Data, Visualization, Friend Connect, or Ajax Language).

Maps/Navigation App Integration

Some platforms, such as iOS, Windows Phone, and Android, include a native map application outside of the browser that we may want to open in some situations, such as to start a turn-by-turn navigation process. To communicate with this app from our website, we will just use standard URIs, as we will see in Chapter 15.

Google Maps for Android

To invoke the Google Maps application on Android devices we can just point to `maps.google.com` and optionally send any parameters we want to include, in the basic form `http://maps.google.com/?<attributes>`. Attributes should be URL-formatted, as in `attribute1=value1&attribute2=value2`. Possible attributes include:

q

Query parameter; this can be a comma-separated coordinate preceded by a *loc:* prefix (*loc:lat, long*), or any search string, such as *starbucks*

near

Applies a location definition for a query, as in *q=starbucks;near=san+mateo+ca*

ll

A comma-separated latitude and longitude for the map center

l

The type of map (*m*: map, *k*: satellite, *h*: hybrid, *p*: terrain)

z

The zoom level, from 1 (the whole world) to 23 (buildings, not available in all areas)

On Android devices, the user will be able to decide if she wants to execute URLs with the Google Maps native API or with the browser.

To force the use of the Maps application, we can also use the *geo*: URI (currently being discussed as a standard; see IETF (<http://bit.ly/Z6uzmz>)). The basic syntax includes:

```
geo:latitude,longitude
geo:latitude,longitude?z=zoom_level
geo:0,0?q=my+street+address
geo:0,0?q=business+near+city
```

Therefore, to make a link to open the Maps application centered on the Eiffel Tower, we can use both options:

```
<a href="http://maps.google.com/?q=eiffel+tower+paris">
  See the Eiffel Tower on a map
</a>

<a href="geo:0,0?q=eiffel+tower+paris">
  See the Eiffel Tower on a map
</a>
```

Directions and navigation

To initiate a route algorithm and later a possible turn-by-turn navigation, we can use <http://maps.google.com> with the *saddr* (source address) and *daddr* (destination address) parameters, such as:

```
<a href="http://maps.google.com/?saddr=golden+gate&daddr=pier+39">
  Directions
</a>
```

Street View

Android devices also offer the Street View service on Google Maps. To open a Street View panorama, we can use the nonstandard *google.streetview*: protocol. The parameters are:

cbll

The latitude and longitude, comma-separated (mandatory)

cbp

A series of optional parameters, such as yaw (center of panorama view in degrees clockwise from north), pitch (center of panorama view in degrees from -90 to 90), and the panorama zoom

mz

The map zoom associated with this panorama

If you want to get these parameters, you can just browse to the street view you want to link to and generate a link from your desktop; you will see the parameters there.

To make a link to the White House street view, we can then use:

```
<a href="google.streetview:cbll=38.900214,-77.036509&cbp=12,187.16,,0,2.74">  
  View the White House</a>
```

iOS Maps

The Maps application on iOS has changed with time. From iOS 1 to iOS 5.1 the application is Google Maps, and from iOS 6 the application is Apple Maps.

For the Google Maps application we can use the same *http:* protocol as in Android, so linking to *http://maps.google.com* will open the Maps application instead of the website until iOS 6. It accepts the same parameters as in Android.



While not officially documented, Apple supports the *maps:* protocol in both iOS 5 and iOS 6, accepting at least the *q* parameter for specifying a query.

From iOS 6, *http://maps.google.com* will point to the HTML5 website; to open the native Maps application we need to replace *google* with *apple*. That is, instead of:

```
<a href="http://maps.google.com/?q=eiffel+tower+paris">  
  See the Eiffel Tower on a map  
</a>
```

we need to use:

```
<a href="http://maps.apple.com/?q=eiffel+tower+paris">  
  See the Eiffel Tower on a map  
</a>
```



Full documentation on the Apple Maps URI scheme can be found at *mobilexweb.com/go/applemapsuri*.

The iOS Maps URI scheme adds to the Google scheme some proprietary attributes, such as *cid* for a business ID in the Apple places database.

Directions and navigation

To initiate a route navigation, we can use the same syntax as in Google Maps from iOS 6. In this case, Apple Maps will calculate the route and offer to the user to start a turn-by-turn navigation (Figure 14-2):

```
<a href="http://maps.apple.com/?saddr=golden+gate&daddr=pier+39">  
  Directions  
</a>
```

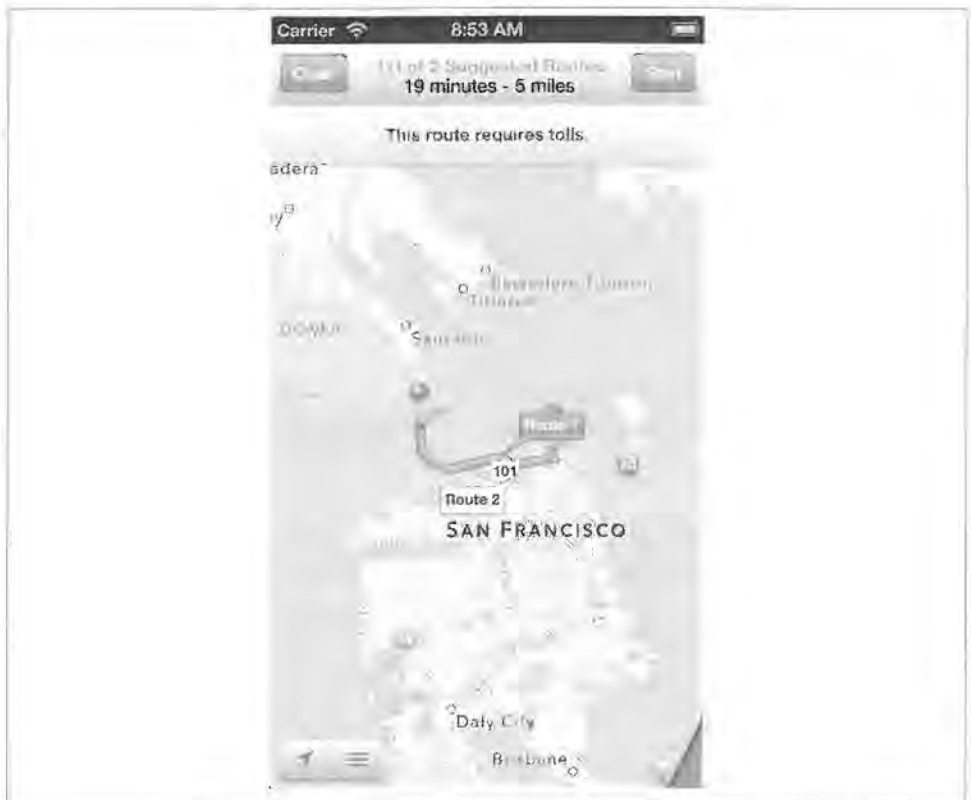


Figure 14-2. From a mobile website we can open the native Maps application, including the ability to start a route generation and a turn-by-turn navigation operation, using anchor elements and some specific URIs.

Leaving *saddr* or *daddr* as a blank value, as in:

```
<a href="http://maps.apple.com/?saddr=&daddr=pier+39">  
    Directions to Pier 39  
</a>
```

causes the user's current location to be automatically filled in as the starting or destination address, respectively.

Bing Maps

On Windows Phone, we can open the Bing Maps application using the *bingmaps:* protocol, which accepts the following parameters (a partial list):

cp

The center point—a latitude and longitude, separated by a tilde ~ character

lvl

The zoom level (1–20)

where

A search query on places, locations, or landmarks

q

A search query on a local business

sty

The map style (*a*: aerial, *r*: roadmap)

trfc

Whether or not traffic information should be included (0: no, 1: yes)

rtp

The route definition, with the source and destination addresses separated by a tilde ~ character; if either the source or the destination is undefined it will make a route from/to the current location



Full documentation on the Bing Maps URI scheme can be found at mobilexweb.com/go/binguri.

Following the same Eiffel Tower example, to open a map showing this location on Windows Phone, we can use:

```
<a href="bingmaps:?where=eiffel+tower+paris">  
    See the Eiffel Tower on a map  
</a>
```

Showing a Map

Once we have located the user (via a client or server solution), we may want to display a map showing the user's position, and/or a list of points of interest or other information superimposed on the map.

To do this, we should use one of the available public maps APIs: Google Maps, Bing Maps from Microsoft, or Nokia Here. We can use all these APIs for iOS and Android devices, as well as other HTML5-compatible devices; Nokia Maps doesn't just work on Nokia browsers.



As we saw in the previous section, Apple has its own Maps API from iOS version 6.0; however, there is no public API for nonnative iOS apps at the time of this writing, so we can't use it from our websites. We can still link to it using the Apple Maps protocol (<http://maps.apple.com>).

There are actually two Google APIs that are useful for mobile browsers: the Google Maps API v3 and the Google Maps Static API. The first one is the same service that we can find in any website using Google Maps. However, it is currently compatible only with iPhone and Android devices; on other devices, this API will not work properly. The Static API will allow us to show a static map compatible with any mobile browser.

Google Maps API v3

If we are sure that the device is HTML5-compatible, we should use the Google Maps API v3, as shown in Figure 14-3.

To use the API, we first need to include the script loader:

```
<script type="text/javascript"
      src="http://maps.google.com/maps/api/js?sensor=true">
```

The sensor value must be true if the device has geolocation support via the operating system (this is the case on both iPhone and Android devices). We should check whether the device is compatible either on the server, as discussed in Chapter 10, or using JavaScript.

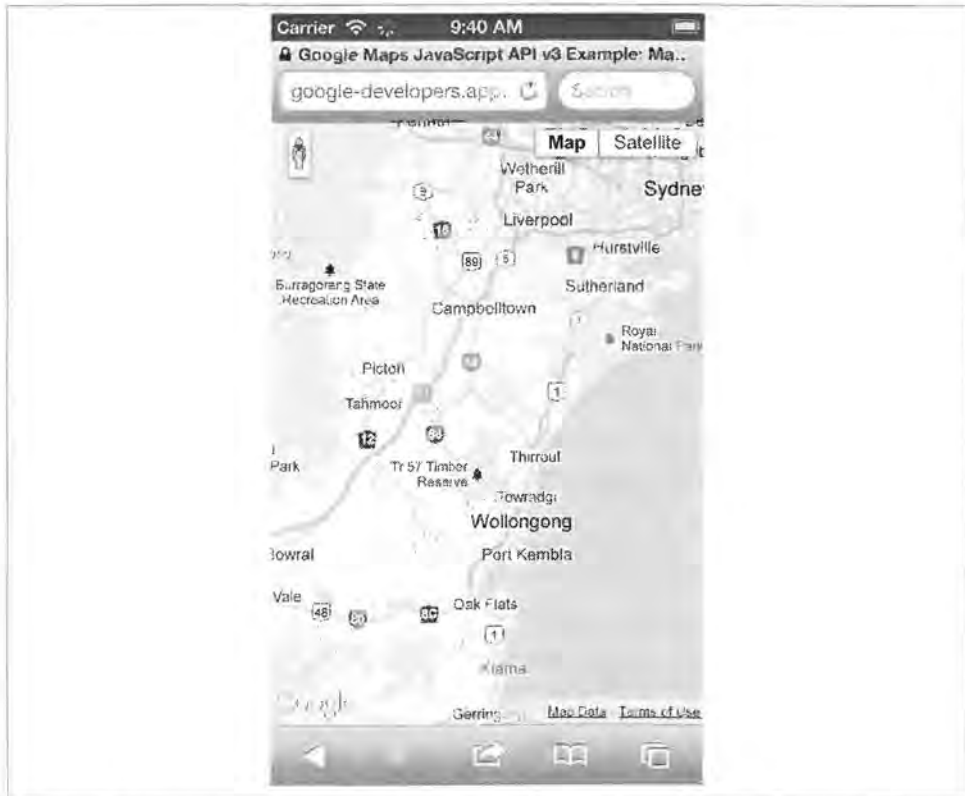


Figure 14-3. Here we can see the Google Maps API on an iPhone showing the user's current location.

Then, we need to create a `div` tag in our HTML and define its dimensions as 100%:

```
function showMap() {
    var useragent = navigator.userAgent;
    var divMap = document.getElementById("map");

    if (useragent.indexOf('iPhone') != -1 ||
        useragent.indexOf('Android') != -1 ||
        useragent.indexOf(iPod) != -1 ) {
        divMap.style.width = '100%';
        divMap.style.height = '100%';
        // ...
    } else {
        // Google Maps not compatible with this mobile device
    }
}
```



On feature phones or cloud-based browsers we must use the Google Maps Static API instead.

The other requirement for mobile devices is to define the meta tag to work without user zooming and with an initial scale of 1.0. This is necessary to avoid usability problems with the map zooming. So, in the head we should add:

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

A full sample looks like this:

```
<html>
<head>
<meta name="viewport" content="initial-scale=1.0, user-scalable=no" />
<script type="text/javascript"
  src="http://maps.google.com/maps/api/js?sensor=true"></script>
<script type="text/javascript">
  function init() {
    var userAgent = navigator.userAgent;
    var divMap = document.getElementById("map");

    if (userAgent.indexOf('iPhone') != -1 ||
        userAgent.indexOf('Android') != -1 ||
        userAgent.indexOf('iPod') != -1 ) {
      divMap.style.width = '100%';
      divMap.style.height = '100%';
      position = getPosition(); // This needs to be implemented
      var latlng = new google.maps.LatLng(position.latitude,
        position.longitude);
      var options = {
        zoom: 7,
        center: latlng,
        mapTypeId: google.maps.MapTypeId.ROADMAP
      };
      var map =
        new google.maps.Map(document.getElementById("divMap"),
          options);
    } else {
      // Google Maps not compatible with this mobile device
    }
  }

</script>
</head>
<body onload="init()">
  <div id="divMap"></div>
</body>
</html>
```

See the full Google Maps API documentation (<http://code.google.com/apis/maps/documentation/v3>).

Google Maps Static API

If we need compatibility with all mobile devices—even the ones without JavaScript support—we can use the free Google Maps Static API, which allows us to show a map as a static image without any automatic interaction.



Up to 25,000 static map requests per application per day are free. If you need more, you need to purchase additional quota through the API website.

This API is very simple and doesn't require any JavaScript or server code. We will use it inside an HTML image tag, in the source URL. The URL will look like this:

```
http://maps.google.com/maps/api/staticmap?parameters
```

To use this API we need to get the location from the server or the client and generate the image URL dynamically, using JavaScript on compatible devices. We need to remember that we may not want to show the user's location, but rather a map of some other place.



With the Google Maps Static API we can show a map on any mobile phone on the market, even those without JavaScript or Ajax support.

The common parameters include:

sensor

Must be true for a mobile device.

center

May be a position using *latitude*, *longitude*, or a city name.

zoom

The level of zoom required, from 0 (world view) to 21 (building view).

size

The size in pixels of the image required (e.g., 220×300). We should get the device's screen size from the server or from JavaScript.

format

Accepts GIF, JPEG, or PNG. The default is PNG, which is suitable for mobile devices.

mobile

Marking this parameter as true creates different rendering images optimized for viewing on mobile devices. Google suggests using false in the case of iPhone or Android devices.



The API is more complex, and it can even show marks and routes over the map. See the full documentation (<https://developers.google.com/maps/documentation/staticmaps/>).

For example, if we are using PHP and we just have the latitude and longitude (acquired by any method), we should use:

```
<?php
// $latitude and $longitude already acquired
$url = "http://maps.google.com/maps/api/staticmap?center=$latitude,
$longitude&zoom=14&size=220x300&sensor=true";
?>

```

Nokia Here

Here (formerly Nokia Maps) offers a series of free services for mapping on websites (<http://developer.here.net>), including:

The Mobile HTML5 Framework

A rich mobile app development framework, including complete functionality for Maps, Positioning, Places, Direction, and Traffic. This API is fully compatible with iOS, Android, MeeGo, Windows Phone, and BlackBerry (see Figure 14-4).

The JavaScript API

A classic JavaScript API that is not necessarily optimized for mobile websites.

REST APIs

Web Services allowing us to access pre-rendered static map images (similar to the Google Static Maps API), a places database, routing and traffic information, and more.

To work with all the APIs, we need to become a Nokia developer (register for free (<http://developer.here.net>)) and then register an API key to use the mapping services (<http://api.developer.nokia.com>). Usage of the mapping APIs is subject to quotas based on the service we are using; enterprise solutions are available.

To use the Mobile HTML5 API (http://developer.here.net/mobile_html5), once we have the API key registered we just need to add some file references, create a map container (usually a div element), and call the API.



jHere (<http://jhere.net>) is a lightweight JavaScript framework that works over Zepto.js or jQuery to simplify the work with Here services, including interactive maps, KML support, and info bubbles.

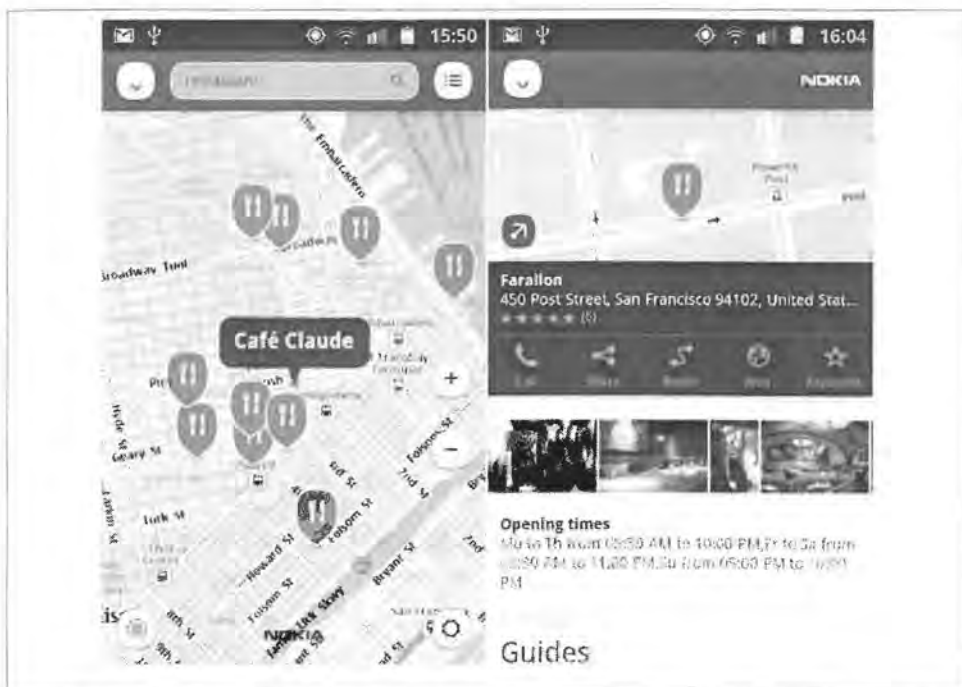


Figure 14-4. Here for mobile HTML5, while created by Nokia, works on different platforms—in this case, an Android browser.

To add the files:

```
<link rel="stylesheet"
      href="http://api.maps.nokia.com/mobile/latest/lib/mh5.css"/>
<link rel="stylesheet"
      href="http://api.maps.nokia.com/mobile/latest/lib/colors.css"/>
<script src="http://api.maps.nokia.com/mobile/latest/lib/mh5.js">
</script>
```

When we already have a div element with specific dimensions defined, we can then create a map reference:

```
nokia.mh5.assetsPath = "http://api.maps.nokia.com/mobile/0.2.0/lib/";
nokia.mh5.app.embed({
  domNode: "#mymap",
  // API credentials
  appId: OUR_APPID,
  appCode: OUR_APPCODE
});
```


CHAPTER 15

Device Interaction

Mobile browsers have the useful facility of being able to interact with the device—both the operating system and the hardware. Communication can be driven by URIs used in links or JavaScript APIs. Some specific platforms support both an API and a URI scheme for the same feature; in that case the API usually is the preferred way as it's more flexible and less error-prone.

Apache Cordova/PhoneGap and some native web platforms add support for more APIs than the ones available in the standard web view on every platform, as we will see later.

Mobile-Specific URIs

There are some URI schemes that many mobile browsers understand that enable us to communicate with some phone features. We can use these URIs in typical `a` elements, and in some browsers we can use JavaScript to force the URI using `location.href`.

Usually we are talking about using a protocol name, a colon, and parameters, such as `tel:2222`:

```
<a href="protocol:parameters"></a>

<script>
  location.href = "protocol:parameters";
</script>
```



The Wireless Telephony Application Interface (WTAI), part of the WAP 1.x standard, was created in the last millennium; the WTAI libraries were preinstalled on many phones and can be accessed by other applications, such as the browser. To use these libraries (if available), we used to use the syntax `wtai://<library name>/<function name>[(;parameter)*]`. Today, only feature phones still support this library.

We need to be careful about using URI schemes that aren't compatible with a device, however, as the user will not understand the error messages. Figure 15-1 shows an example.

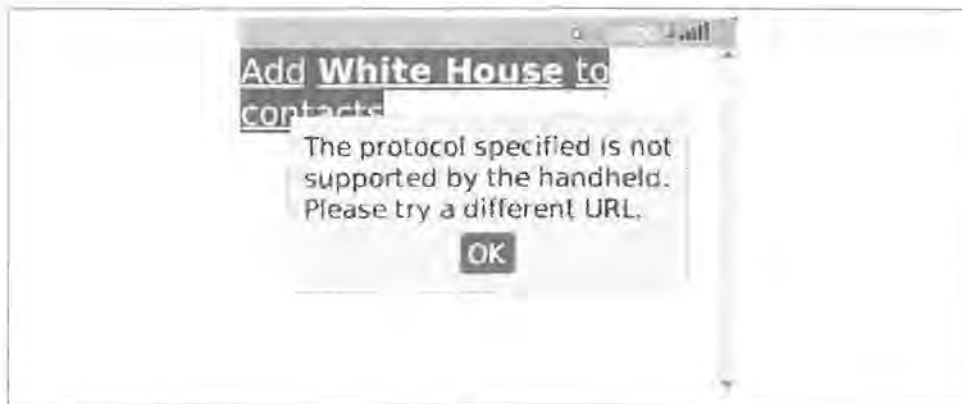


Figure 15-1. Triggering URI schemes that are not compatible with the user's device can lead to strange error messages.

Making a Call

Remember, most mobile devices are also phones! So, why not create link-to-call actions? If you're creating a business guide, or even providing support for your own website, most people will prefer to call a person instead of filling in a form on the device.

Fortunately, there are some URLs that will help us. The first standard (defined by the Internet Assigned Numbers Authority, or IANA) is to use the *tel:**<phone number>* scheme. It's a good idea to provide the phone number in the international format—a plus sign (+), the country code, the local area code, and the local number—as we do not really know where our visitors will be located. For example:

```
<a href="tel:+1800229933">Call us free!</a>
```

If they are in the same country, or even in the same local area, the international format will still work.

Some devices also allow sending DMTF tones after the call has been answered by the destination. This is useful for accessing tone-controlled services, help desk systems, or voicemail; you can say to the link, "call this phone number and, when the call is answered, press 2, wait 2 seconds, and then press 913#." You do this using the *postd* parameter after the number: the syntax is *postd=<numbers>*. You can use numbers, *, and # (using the URL-encoded %23 value), as well as *p* for a one-second pause and *w* for a wait-for-tone pause. For example:

```
<a href="tel:+1800229933;postd=4">Call us free!</a>
```

This function doesn't work on all mobile devices, but on devices that don't understand it, the primary telephone number should at least be called. The compatibility list for this feature is complex, and I don't recommend relying on it.

If the user activates a call link, she will receive a confirmation alert asking whether to place the call, showing the full number so she can decide. This is to avoid fraudsters tricking the user into calling another country or a premium number. As shown in Figure 15-2, some nonphone devices offer a contextual menu allowing the user to copy or save the phone number instead (the same functionality can be achieved on phone devices after a long-press gesture over a link).



Figure 15-2. Some nonphone devices, such as the iPad or the iPod touch, provide the user with a context menu option to copy or save the phone number instead of placing the call.

Not as well supported today as *tel:*, the other way to originate a call is using the WTAI standard, via the *wp* public library and the *mc* (make call) function:

```
<a href="wtai://wp/mc;+1800229933">Call us free!</a>
```

This link can include tones to be sent to the destination as if the user had pressed them on the keypad, specified using the *wp* library's *sd* (send DTMF tones) function.



The BlackBerry browser automatically detects phone numbers and email addresses and converts them to links. If you don't want this feature, you should use the meta tag `<meta http-equiv="x-rim-auto-match" content="none">`. Safari also has its own meta tag for the same action: `<meta name="format-detection" content="telephone=no">`.

iDEN networks (like Nextel) use radio packets to make internal calls inside the network. If you are working with customers of such a network—for example, for an intranet—you can allow users to launch internal calls to other members of the team (or external calls) using the Direct Connect URL scheme (`dc:<number>`). This is compatible with BlackBerry iDEN devices:

```
<a href="dc:5040*0077">Ping John</a>
```



Some models present users with a submenu when they click a *tel:* link so they can choose whether to place a voice-only or a video call (available in 3G systems). Some Japanese phones also allow you to specify that a link should initiate a video call, using the protocol *tel-av:<phone number>*.

Sending Email

Some modern devices with browsers also have mail applications that can react to the classic web *mailto:* protocol. The syntax is *mailto:<email_destination>[?parameters]*. The detected parameters can change from device to device but generally include *cc*, *bcc*, *subject*, and *body*. The parameters are defined in a URL format (*key=value&key=value*), and the values must be URI-encoded.

Here are some samples:

```
<a href="mailto:info@mobilexweb.com">Mail us</a>
<a href="mailto:info@mobilexweb.com?subject=Contact%20from%20mobile">
  Mail us
</a>
<a href="mailto:info@mobilexweb.com?subject=Contact&
  body=This%20is%20the%20body">
  Mail us
</a>
```



Be aware that the *mailto:* mechanism doesn't guarantee that the message will be sent. It generally just opens the mail application, and the user has to confirm the sending after making optional changes. If you need to actually send the mail, use a server mechanism.

Generally, if we want to insert a newline in the body of the email, we can use the Carriage Return plus Line Feed characters (%0D%0A). This does not work with the Mail application in iOS at the time of this writing; however, it does allow us to insert HTML tags inside the body, so we can use `<br>` for Safari on iOS (as well as creating complex HTML messages):

```
<a href="mailto:info@mobilexweb.com?subject=Contact&
    body=This%20is%20the%20body%0D%0AThis%20is%20a%20new%20line">
    Mail us
</a>

<a href="mailto:info@mobilexweb.com?subject=Contact&
    body=This%20is%20the%20body<br />This%20is%20a%20new%20line">
    Mail us from iPhone
</a>
```



The Mail application in iOS supports HTML in the message body, so we can read an HTML content file using XMLHttpRequest or just the innerHTML part of the current page, encode it as a URL, and send it inside the *mailto:* protocol. This enables us to send rich postcards from our websites.

Sending an SMS

We all like the Short Message Service; that's why mobile browsers generally offer the ability to invoke the new SMS window from a link. To do this, we have two possible URI schemes, *sms://* and *smsto://*. Unfortunately, there is no standard way to know for sure which one is compatible with a user's browser.



WURFL supports a capability to query about which SMS URI scheme is supported on the current device. The capability is `xhtml_send_sms_string` and the possible values are `none`, `sms:`, or `smsto:`.

The syntax is *sms[to:][/<destination number>][?parameters]*. As you can see, the destination number is optional, so you can open the SMS composer from the device without any parameters defined. The *parameters* usually define the body, but this

property is not compatible with all phones for security reasons (to avoid a website sending premium SMS texts, for example). As with an email, an SMS is not automatically sent when the user presses the link. The link only opens the SMS composer window; the user must finish the process manually.

On iOS, from version 5.0, the `sms://` protocol also works on the iPad and iPod touch; the user is redirected to iMessages instead of the SMS application, which is not available.

The destination number should be an international number or, if it is a short number code, we should guarantee that the user is in the right country and is connected with one of the compatible carriers of that short code.



The BlackBerry platform supports direct messaging between two BlackBerry devices using BBM (BlackBerry Messenger). To create a direct message, you can use the `pin:<number>` URL scheme.

Here are some samples:

```
<a href="sms://">Send an SMS</a>

<a href="sms://?body=Visit%20us%20at%20http://mobilexweb.com">
  Invite a friend by SMS
</a>

<a href="sms://+3490322111">Contact us by SMS</a>

<a href="sms://+3490322111?body=Interested%20in%20Product%20AA2">
  More info for product AA2
</a>
```

Working with MMS

The Multimedia Messaging Service (MMS) is the standard way to send messages with multimedia content (images, video, or any attached content). It depends on the SMS standards and a content file (the multimedia message) that the sender uploads to the carrier and the recipient downloads from the carrier. Symbian devices allow us to define the URI schemes `mms://<url>` to download an MMS file from the specified URL and `mmsto://<destination number>` to open the Multimedia Message Composer.

Other Communication Apps

There are some native applications that can be opened from a link, mostly on iOS. One example is the *FaceTime* video-chat application for the iPhone, iPad, and iPod touch.

From iOS 4.1 we can use the *facetime://* URI to open the FaceTime application and call somebody. For example:

```
<a href="facetime://5555555555">Call us free using FaceTime!</a>
```

Skype is another application with its own URI scheme, that currently works on iOS and Android. Providing the Skype username with a *?call* parameter will immediately initiate a call when the user clicks the link; without it, clicking the link will bring up the identified user's profile instead. For example:

```
<a href="skype:skype_user">Our Skype profile</a>
<a href="skype:skype_user?call">Call us using Skype!</a>
```



Skype is one of the only applications with good documentation (<https://developer.skype.com/skype-uri>) on how to use its URI scheme on different mobile operating systems.

Google Voice on Android will use the standard *tel:* and *sms:* protocols if the user selects it as the application for those protocols. WhatsApp on Android uses a similar mechanism for messaging; it has no specific URI.

Adding a Contact to the Phonebook

It might be useful to invite users to add your company's contact information (only the phone number, or full details) to their phonebooks for future communication. A WTAI function is available for this purpose for older and WML-compatible devices, and there's also a tricky way of doing it for some modern (and smarter?) devices.

The WTAI library is *wp*, as for making phone calls, and the function is *ap* (add to phonebook). The parameters are the number and optionally a name to be assigned to it, separated by a semicolon. For example:

```
<a href="wtai://wp/ap;+12024561111;White%20House">
  Add <strong>White House</strong> to contacts
</a>
```

For modern browsers (those that don't support WTAI—basically, every modern smartphone or tablet) the trick is to create a *vCard* file (vCard is a standard file format for electronic business cards). If you link to this file, most browsers will send the file to the device's Phonebook application, and the user will be invited to add the contact to the database. Later we will also see some APIs that will help us with this matter.

A simple vCard 2.1 file (the most compatible version for mobile devices) will look like this:

```
BEGIN:VCARD
VERSION:2.1
```

```
N:Maximiliano;Firtman
ORG:O'Reilly Media
TITLE:Author
TEL;CELL;VOICE:+133MFIRTMAN
TEL;WORK;VOICE:+541150320077
END:VCARD
```

For the device to detect this text file as a valid vCard, we must deliver it with the MIME type `text/x-vcard`. The file, if static, is generally a `.vcf` file.

Unfortunately, many modern (again, smarter?) mobile browsers, like Safari on iOS, don't understand vCards if you provide them as links in a document. However, they do understand them if the user receives them by email! The solution, therefore, is to retrieve the user's email address and, from the server, send the vCard as an attachment.

Another excellent feature to provide to compatible devices is to automatically add to the user's calendar details about a meeting or appointment scheduled online, or a reminder about an event for which the user has bought tickets. If you're selling tickets to a concert or a theater show, this is a great way to ensure that the user won't forget it.

There isn't a WTAI way to do this—in fact, in 1998 mobile devices generally didn't have calendar programs. For compatible devices we can use the *iCalendar* format, based on an older vCalendar standard similar to vCard. iCalendar files should be served with the MIME type `text/calendar`.



Safari on iOS has the ability to show PDF, RTF, Microsoft Office, and Apple iWork files just by linking to them, without the need of any app. Other platforms usually require an app capable of viewing those kinds of files.

Integrating with Other Applications

Some devices allow us to integrate our websites with other native installed applications. This is dramatically nonstandard, though, and it depends very much on the device and the applications that the user has installed.

iOS URL schemes

Safari supports some standard URL schemes that will open other native applications. For example, we can open the Maps application simply by using a classic URL pointing to `maps.apple.com`, as we saw in Chapter 14. For YouTube, we can also make `http://` links to `youtube.com` and they will be redirected to the YouTube app (both the default app available up to iOS 5.1 and Google's app from the App Store, available on demand from version 6.0).

From iOS 6 we can take advantage of *Smart App Banners* to communicate with a native app. In this case we are not in control of how the banners are shown, and the operating system decides where to show an installation invitation or an open invitation. We covered Smart App Banners and how to set them up in Chapter 7.



Up to iOS 5.1, we could use `maps://` or `http://maps.google.com` to open the native Maps application. Starting with iOS 6, Google Maps is no longer available as a native app and has been replaced by Apple Maps. We can communicate with this version using `maps://` or `http://maps.apple.com`.

An updated list of URL schemes for iOS is maintained by Akosma (http://wiki.akosma.com/IPhone_URL_Schemes), and a list of applications, URL schemes, and optional parameters for Safari can be found at Handle Open URL (<http://www.handleopenurl.com>).

For example, according to these sites we can open the Facebook iOS native application (if installed) showing the user's list of friends using the URL `fb://friends`, and if the user has installed Twitter, we can post a message directly using `twitter://post?message=<msg>`. Here's a sample:

```
<a href="twitter://post?message=I%20have%20just%20visit%20a%20site"> Tweet this post
</a>
```



When passing parameters by URL, remember to properly encode spaces and other characters (for example, using `%20` for spaces).

Detecting Whether an Application Is Installed on iOS

When the user clicks a link in mobile Safari that is designed to open another application, what happens if that application is not installed on the user's device? In this case, the browser shows an alert to the user and continues displaying the same page. We can take advantage of this behavior to improve the user experience. We can create a timer in JavaScript that starts when the user clicks the link; if the timer is executed quickly, it means that the application could not be opened because it is not installed.

We can't avoid the Safari message, but we can show the user an error message indicating that the application seems not to be installed, while providing a link to the App Store so the user can buy or download it. If we redirect the user to another website or to the App Store itself, the error message will be on the screen for just a couple of milliseconds and it will be replaced automatically by the next call.

Also remember that from iOS 6 we have the ability to show Smart App Banners, as we covered in Chapter 7.

Using these techniques, there are free and commercial services that provide URL shorteners to use in social networks that will try to open the native app, or the store if the app is not installed. mURL (<http://www.murl.cc>) is one of these services.

X-Callback-URL. When you open a native app from a link, your website is minimized or closed as in iOS only one app can be opened as the foreground app. Therefore, after doing something with the app, the user needs to come back to the website manually.



Android and Windows Phone have a Back button that will take the user back to your website from a native app after opening it via a URI mechanism.

To solve that problem, Greg Pierce has developed a specification that some apps are adhering to, called X-Callback-URL (<http://x-callback-url.com>). The basic idea is that we can add some specific parameters to the URL that the app will use as a callback mechanism to return to our website when the operation carried out in the native app is finished. While this mechanism was created for app-to-app communication, it can also be used by websites. The only drawback is that when the user goes back it may open the callback in a new window/tab, leaving the caller in the background.



On iOS at the time of this writing there is no way to replace the *http://* protocol, and Safari will always open these links. Therefore, Chrome for iOS has registered the *googlechrome://* and *googlechromes://* (for secure connections) protocols, which will force opening a website in Chrome if it's installed.

Android intents

Android also has the ability to communicate with other applications, via *intents* (abstract descriptions of actions to be performed). A native Android application can register an intent as an explicit call (not available from web applications), from a URL, or from a MIME type.

For example, we can open the default PDF viewer on the user's device by delivering a PDF file with the MIME type *application/pdf*. To open an application without sending a file, we can use the URI schemes defined by the intent. For example, linking to a YouTube video will fire the YouTube internal application (if the user has defined it as the default player).

When the intention call is implicit (using a URL instead of a unique package name), there may be more than one installed application that can respond to that URL. In this case, the user will receive a pop-up asking her to select the application to use. The user can also select a default application for future usages.



In Android, an intent can be registered as part of an HTTP URL or be activated from a MIME type. For example, if we link to <http://www.google.com/m/products/scan> in the Android browser, it will try to open a barcode scanner application.

Among the internal URLs that Android supports are those for Google Maps placemarks using `geo:<latitude>,<longitude>`, Google Maps searches using `geo:0,0?q=<search>`, and Google Street Views using `google.streetview:cbll=<latitude>,<longitude>&cbp=1`, as we covered in Chapter 14. More information is available at Android's developer site (<http://developer.android.com/guide/appendix/g-app-intents.html>).



In Android, when a URI scheme is not registered as an intent, that means the application is not installed. The user will be taken to a new page showing an error message, like when you link to a website that doesn't exist.

Unfortunately, at the time of this writing there are no websites that list all the possible URI schemes to use on Android. Usually well-known applications, such as Facebook or Twitter, use the same URI scheme as on iOS when available.

Symbian local applications

Symbian devices also allow us to open applications using the nonstandard *localapp:* scheme. For example, to open the Calendar, Contacts, or Messages app, we can use *localapp:calendar*, *localapp:contacts*, or *localapp:messaging*, respectively.

Windows apps

Windows 8 and Windows Phone from version 8.0 support URI association from native apps. Windows 8 includes some built-in app URIs that are not documented at the time of this writing, such as *maps*, *ms-excel*, *ms-word*, and *wallet*, as well as URI schemes to access the Settings app.

For Windows Phone 8, Microsoft offers a list of URIs (<http://www.mobilexweb.com/go/wp8uri>) for accessing specific sections of the Settings app and the Windows Phone Store.

JavaScript APIs

Most device interaction can be achieved through JavaScript APIs. Some of these APIs are specs from the W3C, some are just spec drafts from other sources, and others are proprietary APIs for one specific vendor.



Data, network, geolocation, drawing, and multimedia APIs are covered in different chapters in this book.

Touch

Most tablets and smartphones today are touch devices, and some of them also have some kind of a mouse pointer. For example, the Microsoft Surface allows you to use an external mouse or the touch keyboard.

Touches are different than mouse clicks in two main areas:

Click area

A touch area is larger than a click area (several pixels as opposed to just one). Even if the user touches quickly, the pressure center may change, and as a mouse event it can be detected as a mouse move.

Multiplicity

Modern devices are multitouch, meaning that we may detect more than one touch at the same time. Standard mouse events don't support multiplicity.

While every device still emulates mouse events when touches are used, there are some specific touch events that we can use too. With touch we can also detect gestures manually, or with APIs available in some browsers.

In terms of implementations, today we can find two: Apple's touch and gesture events, and Microsoft's pointer and gesture events.

Before we start talking about the APIs, it's important to understand that some browsers, and some operating systems, have their own touch gestures, and we should avoid trying to detect them at the same time. The following is a list of some gestures to avoid:

- On Safari for iPad, the OS detects four-finger gestures for going home, switching between apps, and accessing the previously opened app list. Therefore, we should avoid four-finger gesture detection.
- Chrome for mobile devices and tablets supports gestures to move between windows or tabs. On phone-factor devices it's a one-finger swipe (or flick) gesture starting from the edge of the window (left or right). Therefore, if we want to use swipe gestures, we should avoid detecting such gestures if they start in the leftmost or

rightmost 50 CSS pixels of our page. A visual indicator for the user to know that she should start the gesture inside that area will help.

- MeeGo supports gestures from the borders of the screen to move between open apps, and in some other situations. Therefore, in MeeGo browsers (the Nokia browser, Opera, or Firefox) we should avoid detecting gestures over the edges. The same thing goes for BlackBerry PlayBook and BlackBerry 10 devices.
- Internet Explorer 10 in Windows 8-style mode (also known as Metro mode) supports swipe events (starting anywhere on the screen) to move backward and forward in history.

Apple Touch Events API

As the first multitouch browser, Safari on iOS invented the touch API as a proprietary extension with the following nonstandard events:

- `touchstart`
- `touchmove`
- `touchend`
- `touchcancel`

After a while, other browsers—including the Android browser, Chrome, the BlackBerry browser, Opera, and Firefox—cloned Apple's implementation.



While some devices support multitouch gestures, the browser may not support them—for example, the Samsung Galaxy Note I exposes the Touch Events API but its browser supports only single-touch gestures.

When we capture these events, we will receive them both for single touches and multitouches. Every time the user presses a finger on the screen, `touchstart` will be executed; if he moves one or more fingers, `touchmove` will be the event to capture; and when the user removes his finger(s), `touchend` will be fired. What about `touchcancel`? A touch cancel event is executed if any external event with more priority than our website (such as an alert window, an incoming call, or a push notification) cancels the operation. Unfortunately, the `touchcancel` event has several bugs in terms of how it's implemented in different browsers.



If you are creating a game, a drawing application, or some other solution capturing touches, it is very important to remember the `touchcancel` event and to pause or stop the touch behavior when this event fires.

The four events receive the same event object (TouchEvent) as a parameter. It contains a `touches` collection representing each touch associated with the current event and a `changedTouches` collection representing the touches that have changed since the last call. Each collection element is a Touch object having primarily coordinate properties, such as `pageX` and `pageY` properties.



We can bind touch events using either `addEventListener` or the `on<event_name>` DOM property for either the whole document or any particular element on the page.

A typical scenario, then, will be:

```
<div
  ontouchstart="touchStart(event);"
  ontouchmove="touchMove(event);"
  ontouchend="touchEnd(event);"
  ontouchcancel="touchCancel(event);">
</div>
```



A touch sequence begins with the first finger and ends with the last finger. The touch events will be delivered to the same object that received the `ontouchstart` event, no matter where the current touches are located.

We may want to cancel in all events the default behavior of Safari for the gesture the user is doing. This can be done with the `preventDefault()` method of the TouchEvent object:

```
event.preventDefault();
```

The TouchEvent object supports the array collections shown in Table 15-1.

Table 15-1. TouchEvent collections

TouchEvent attribute	Description
<code>touches</code>	All the touches actually on the screen
<code>targetTouches</code>	Only the touches inside the target element of the event
<code>changedTouches</code>	Only the touches that have changed since the last event call (useful in <code>ontouchmove</code> and <code>ontouchend</code> or to filter only new or removed touches)

When the user lifts a finger from the screen, that touch will be available in `changedTouches` but not in the other collections. In Android 2.x, however, the removed touch is also available in the `touches` collection—you should test your code on every browser, as you may find differences.

Every Touch object has the properties outlined in Table 15-2.

Table 15-2. Properties of the Touch object

Touch attribute	Description
clientX, clientY	Touch coordinates relative to the viewport
screenX, screenY	Touch coordinates relative to the screen
pageX, pageY	Touch coordinates relative to the whole page, including the scroll position
identifier	A number for identifying the touch between event calls
target	The HTML element where the event was originated

The following sample will show a blue 20px circle below each finger touching the screen:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>iPhone Multitouch</title>
<meta name="viewport" content="width=device-width; initial-scale=1.0;
    maximum-scale=1.0; user-scalable=0;">
<style>
    .point {
        width: 20px;
        height: 20px;
        position: absolute;
        -webkit-border-radius: 10px;
        background-color: blue;
    }

</style>
<script>
function touch(event) {
    event.preventDefault();
    for (var i=0; i<event.touches.length; i++) {
        var top = event.touches[i].pageY-10;
        var left = event.touches[i].pageX-10;
        var html = "<div class='point' style='left: " + left +
            "px ; top: " + top + "px'></div>";

        document.getElementById("container").innerHTML += html;
    }
}

function clean() {
    document.getElementById("container").innerHTML = "";
}

</script>
</head>
```

```

<body>

<div ontouchstart="touch(event)" ontouchend="clean()" id="container"
    style="background-color:red; width: 300px; height: 300px">

</div>

</body>
</html>

```

Disabling scrolling. We can use touch events to disable scrolling on our website, which can be useful for immersive apps or games where we don't want the content to be scrolled. Even when there is no overflow content, Safari on iOS allows a momentum scroll where you can even see the background behind the web page.

To disable scrolling, we can just use the following snippet of code:

```

document.addEventListener("touchmove", function(event) {
    event.preventDefault();
}, false);

```

BlackBerry touch behavior. The BlackBerry browser since version 7 (including the PlayBook) supports a meta tag to define how the browser should react to touch and cursor events raised by the touchpad or a mouse connected to the PlayBook.

The `touch-event-mode` meta tag defines how the browser handles touch events, and if it should detect browser gestures (such as click, scroll, or zoom) or pass all the raw touch information to the website. The possible values are `native` (no browser reaction to touches, even basic behavior), `pure-with-mouse-conversion` (only basic touches are mapped to click events), and `processed` (default value, the browser will manage all touch gestures).

For example, if we are creating a game and we don't want the browser to handle touch interactions, we can use:

```

<meta name="touch-event-mode" content="native">

```

A similar meta tag is available through `cursor-event-mode`, accepting the values `native` (the browser will not handle double-click to zoom or click and hold for a menu) or `processed` (default value).

W3C Touch Event API

After different vendors started to support the Apple Touch Events API (<http://www.w3.org/TR/touch-events>), the W3C started a working group to standardize it. The BlackBerry browser was the first browser to follow the W3C spec.

The W3C Touch Event API is a superset of the Apple API, adding the `touchenter` and `touchleave` events for dragging purposes. If we are listening for those events on a

specific DOM element, `touchenter` will be fired if the user has started a touch outside our element but then drags her finger inside it, and `touchleave` will be fired when a touch that originates in our element leaves it.



While the W3C has reached a recommendation spec, its implementation for the future is still unclear as Apple has filed a patent claim that may conflict with these events. The discussion is still up in the air, and the last W3C recommendation (<http://www.w3.org/2012/te-pag/pagreport.html>) indicates that Apple's patents don't affect this specification. However, Microsoft has decided to use a different implementation, as we'll see shortly. Apple has also decided to not participate in the W3C Touch Events group.

Compatibility

The number of simultaneous touches that we can detect may vary per browser and per device. Table 15-3 shows current compatibility with touch and multitouch events as well as support for the mentioned APIs.

Table 15-3. Browser compatibility with Touch Events APIs and multitouch support

Browser/platform	API support	Max simultaneous touches
Safari on iOS	Apple Touch Events	5 on iPhone and iPod touch 11 on iPad
Android browser	Apple Touch Events	1–10, depending on device
Chrome for Android	Apple Touch Events	1–10, depending on device
IE9 for Windows Phone	No support	N/A
IE10 for Windows Phone	Only MS pointer events	2–4
Symbian browser	Apple Touch Events from version 8.2 (no touch cancel support)	Depends on the device
Nokia Browser for Series 40	No support	N/A
Firefox	W3C Touch Events since 18	Depends on the device
BlackBerry browser for smartphones	W3C Touch Events since 7.x	4
BlackBerry browser for PlayBook and BB10	W3C Touch Events	10
UC browser	Apple Touch Events	Depends on the device
Opera Mobile	Apple Touch Events	Depends on the device
Opera Mini	No support	N/A
Amazon Silk	Apple Touch Events	8

Microsoft pointer events

Microsoft arrived late to HTML5, and to touch events in particular. However, that delay allowed Microsoft to be one of the first to realize that Apple has patented the touch

events, and that's why (from Internet Explorer 10) it is using its own variation of these events, known as *pointer events*, for touch and stylus detection.

While at the time of this writing the only implementation is the Microsoft prefixed one, Microsoft has proposed it to the W3C as a patent-free standard for touch events; the group has been accepted and is in discussion at W3C's website (<http://www.w3.org/2012/pointerevents>).

The main differences in pointer events are that it is input-type-agnostic, supporting touch, mouse, and stylus touches in one API, and that it supports hover (over and out), pressure, and tilt values for compatible pointer types.

The events available in the Pointer Events specification are listed in Table 15-4. Remember that Internet Explorer 10 on Windows 8 and Windows Phone since version 8 supports these events with the MS prefix; therefore, `pointerdown` becomes `MSpointerdown`.

Table 15-4. Available pointer events

Event	Description
<code>pointerdown</code>	The pointer enters the active state.
<code>pointerup</code>	The pointer leaves the active state.
<code>pointercancel</code>	A hardware or operating system operation has canceled the user's pointer action, or a browser gesture has been detected after <code>pointerdown</code> , such as a pinch-to-zoom operation. This can also be triggered if the hardware believes that the touch was accidental or if the user is changing the device's orientation while the pointer is active.
<code>pointermove</code>	The pointer has changed coordinates, pressure, or tilt (if supported).
<code>pointerover</code>	A pointer has been detected in the area but the pointer is not active yet. For noncompatible input types, <code>pointerover</code> should be fired immediately before <code>pointerdown</code> .
<code>pointerout</code>	After a <code>pointerover</code> , the pointer has left the area.
<code>gotpointercapture</code>	The element has captured a pointer.
<code>lostpointercapture</code>	The element has lost a pointer capture.



We can capture pointer events using `addEventListener` or through the DOM interface, which creates one attribute per event following the `on<event>` syntax (such as `onpointerdown` for the standard spec and `onmspointerdown` for IE's implementation).

Pointer detection. The Pointer Events specification adds two properties to `window.navigator`: `pointerEnabled`, a Boolean value to detect API support, and `maxTouchPoints`, an integer value for detecting the maximum number of simultaneous touches

supported by the device. IE10 supports this prefixed, so the properties are `navigator.msPointerEnabled` and `navigator.msMaxTouchPoints`.

Touch action style. The Pointer Events specification adds a CSS style to define the default touch action in a specific area: the `touch-action` style, which accepts the values `none`, `auto`, and `inherit`. By defining `touch-action: none` we can disable the default zooming and scrolling behavior in an area without any JavaScript hack.

In Internet Explorer 10 this property is prefixed (`-ms-touch-action`) and it also accepts more values, such as `pan-x`, `pan-y`, `pinch-zoom`, `manipulation`, and `double-tap-zoom`. Therefore, to disable scrolling and zooming in an element in IE10, the code looks like:

```
#element {  
  -ms-touch-action: none;  
}
```



IE10 for Windows Store apps supports several CSS extensions to manipulate zooming and scrolling behavior, such as `-ms-content-zoom`, `-ms-scroll-limit`, and `-ms-scroll-rails`.

Working with the touches and clicks. All pointer events, such as `MSPointerDown`, receive the same `PointerEvent` object as the event argument. Pointer events differ from touch events in how we receive each touch or point: instead of receiving an array of touches (as in touch events), in pointer events we just receive the current pointer. If the user is pressing with more than one finger simultaneously, we receive one call to the handler per touch, and through the `pointerId` property we can detect if the pointer is the same as or different than the one in the previous calls.

The `PointerEvent` object inherits from `MouseEvent`, including well-known properties such as `clientX` and `clientY` for coordinates, and it adds the properties listed in Table 15-5.

Table 15-5. Most useful `PointerEvent` properties

Property	Description
<code>currentPoint</code>	Gets a <code>PointerObject</code> object with basic info for the pointer.
<code>hwTimestamp</code>	The time at which the pointer was detected by the hardware.
<code>intermediatePoints</code>	A collection of <code>PointerObject</code> objects representing the pointer history.
<code>isPrimary</code>	Returns whether this pointer is the primary one for the current interaction. In a multitouch operation, the first touch is declared as the primary contact.
<code>pointerId</code>	A unique identifier for the current touch or click. It can be used to detect if subsequent events are triggered from the same pointer.

Property	Description
<code>pointerType</code>	The type of the pointer. Possible values are the constant values of the same <code>PointerEvent</code> object: <code>POINTER_TYPE_TOUCH</code> , <code>POINTER_TYPE_PEN</code> , and <code>POINTER_TYPE_MOUSE</code> . In IE10, the constants are prefixed, as in <code>MSPOINTER_TYPE_TOUCH</code> .
<code>pressure</code>	The pressure of the current pointer, from 0 to 255. Available only on selected devices and input types.
<code>tiltX</code>	The angle between the Y-Z plane, in the range of -90 to $+90$. If the angle is greater than 0, that means the tilt is to the right. Available only on selected devices and input types.
<code>tiltY</code>	The angle between the Y-Z and transducer-X planes, in the range of -90 to $+90$. If the angle is greater than 0, that means the tilt is toward the user. Available only on selected devices and input types.
<code>timeStamp</code>	The time at which the pointer event was fired.



While event names, constants, and CSS styles are prefixed in Internet Explorer, pointer properties, such as `pointerId`, are not prefixed.

Simulating touch on desktops. Google Chrome for desktops supports touch event simulation through the developer tools, as we will see in Chapter 18. With the option in active mode, mouse events will also trigger single touch events on desktops.

Touché (<https://github.com/davidcalhoun/touche>) is an open source JavaScript library that we can use while developing that will map mouse events to Apple touch events. *Phantom Limb* (<https://github.com/brian-c/phantom-limb>) offers the same functionality with the addition of simulating a second touch using accessory keys on the keyboard.

Gestures

A *gesture* is a way of combining finger movements over the screen to fire an action, instead of using a simple touch or click. A complete touch (or mouse) move-capturing feature is required in order for gestures to be registered. While we can use low-level touch events and some algorithms to detect them, some browsers support native APIs that will help us with this work.

If the users need to use a gesture in your web application, it is important to train them in what to do by showing a help message, a sample animation, or some other kind of hint.

Swipe gesture

The swipe (also known as *flick*) gesture is a touch-based browser technique typically used for going forward and backward. For example, it is used in many photo galleries to change the currently displayed image, and in presentations to move from slide to slide. The gesture is simply a finger moving across the *x*-axis from left to right or right

to left (a horizontal swipe), or along the *y*-axis from top to bottom or bottom to top (a vertical swipe). It is a one-finger gesture, so it is compatible with almost any touch device. There is no standard event that captures the swipe action, so we need to emulate it using standard events.

The steps will be:

1. Capture `mousedown` (or `touchstart` for compatible browsers) and start a gesture recording.
2. Capture `mousemove` (or `touchmove` for compatible browsers) and continue the gesture recording if the move is on the *x*-axis (or *y*-axis), within a certain threshold. Cancel the gesture if the move is on the other axis.
3. Capture `mouseup` (or `touchend` for compatible browsers) and, if the gesture was active and the difference between the original and final coordinates is greater than a predefined constant, define a swipe to one direction.

The last item can be replaced with an on-the-fly verification of the gesture inside the `mousemove` event.



If you are interested in gesture diagrams, Luke Wroblewski has a complete guide (<http://www.lukew.com/ff/entry.asp?1073>).

Frameworks

Reinventing the wheel is not always a good idea—sometimes it is, but creating algorithms for gesture detection using raw touch movements isn't one of those times! Many application or UI frameworks, such as jQuery Mobile, Dojo Mobile, and Sencha Touch, support some gesture detection in an easier way. There are also some frameworks specifically designed for gesture interactions, such as Hammer.js.

Hammer.js (<http://eightmedia.github.com/hammer.js>) is an open source JavaScript library for multitouch gestures that is available in a compressed format (2 KB). It can detect tap, double-tap, swipe, tap-and-hold, transform, and drag gestures.

Swiper (<http://www.idangero.us/sliders/swiper/>) is a lightweight and multiplatform solution to create swipe effects and transitions using touch or pointer events and CSS transitions, by default.

Other frameworks with gesture support include Jo (<http://joapp.com>), jGestures for jQuery (<http://jgestures.codeplex.com>), Zepto.js (<http://zeptojs.com>), and Lungo (<http://lungo.tapquo.com>).

Apple Gesture API

One of the coolest features of the iPhone when it was presented as a new phone was the zoom and rotate gesture. Using a pinching gesture with two fingers, the user can zoom in and zoom out on content (generally a picture), and using two fingers moving in a circle, he can rotate that picture.



For nonmultitouch devices, we should provide zoom features using normal floating buttons or with a slider.

Fortunately, from iOS 2.0, Safari allows us to detect these gestures without using low-level math in the touch events. This specification is known as Apple Gesture API. While lots of vendors have cloned the Apple touch events, at the time of this writing only the Nokia Browser for MeeGo has implemented this Gesture API.

A gesture, from this API's point of view, is a two-finger movement intended to zoom or rotate the document or a specific element on the screen.



The Gesture API works only with zoom and rotate; it doesn't allow us to detect swipe, touch-and-hold, or any other kinds of gestures.

There are three events available, listed in Table 15-6.

Table 15-6. Events available in the Apple Gesture API

Event	Description
gesturestart	Fired when the user starts a gesture using two fingers
gesturechange	Fired when the user is moving her fingers (rotating or pinching)
gestureend	Fired when the user lifts one or both fingers

The same events are used for rotate and zoom gestures. All three events receive a `GestureEvent` parameter. This parameter has typical event properties, and the additional properties `scale` and `rotation`.

The `scale` property defines the distance between the two fingers as a floating-point multiplier of the initial distance when the gesture started. If this value is greater than 1.0 it is a pinch open (zoom in), and if it is lower than 1.0 it is a pinch close (zoom out).



The Gesture API for iOS works not only in Safari but in every web view, such as an Apache Cordova/PhoneGap app or any web view-based browsers, such as Chrome for iOS.

The `rotation` value gives the delta rotation from the initial point, in degrees. If the user is rotating clockwise we will get a positive degree value, and we'll get a negative value for a counter-clockwise rotation.

I know what you're thinking right now: "Great! Rotation and zoom. But we're working in HTML, so what we can do with that?" CSS3 comes to our help with one attribute, `transform`, and two functions available for manipulating its value: `rotate()` and `scale()`. Remember that at the time of this writing, `transform` is a prefixed attribute in Safari on iOS.

The `rotate()` function receives a parameter in degrees, and we need to define the `deg` unit after the number (for example, `rotate(90deg)`). We can define it from a script using `element.style.webkitTransform`.

Let's look at a simple sample:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Gesture Management</title>
<meta name="viewport" content="width=device-width; initial-scale=1.0;
    maximum-scale=1.0; user-scalable=0;">

<script type="text/javascript">
function gesture(event) {
    // We round values with two decimals
    event.target.innerHTML = "Rotation: "
        + Math.round(event.rotation*100)/100
        + " Scale: " + Math.round(event.scale*100)/100;
    // We apply the transform functions to the element
    event.target.style.webkitTransform =
        "rotate(" + event.rotation%360 + "deg)"
        + " scale(" + event.scale + ")";
}

</script>

</head>

<body>

<div ongesturechange="gesture(event)" style="background-color:silver;
    width: 300px; height: 300px">
```

```

</div>

</body>
</html>

```

Users can rotate and scale the `div` (with all its contents) using two fingers on compatible devices. What's the only problem? The `transform` style is always applied to the original element. So, if we apply a scale of 2.0 to the element and later apply a second scale of 0.5, the new scale value will be 0.5 and not 1.0, as we might expect.

For typical zoom-rotate relative behavior, we should change our function to the following:

```

<script type="text/javascript">
var rotation = 0;
var scale = 1;

function gesture(event) {
    event.target.innerHTML = "Rotation: " +
        Math.round((event.rotation+rotation)*100)/100
        + " Scale: " + Math.round((event.scale*scale)*100)/100;
    event.target.style.webkitTransform =
        "rotate(" +
        (event.rotation+rotation)%360 + "deg)"
        + " scale(" + event.scale*scale + ")";
}

function gestureend(event) {
    rotation = event.rotation+rotation;
    scale = event.scale*scale;
}

</script>

</head>

<body>

<div ongesturechange="gesture(event)" ongestureend="gestureend(event)"
    style="background-color:silver; width: 100%; height: 300px">

</div>

```



If you have a viewport definition allowing zooming actions, remember to prevent the default behavior when the gesture starts so you can manage the gesture, and not the browser.

Microsoft Gesture API

Along with its alternative to the Touch Events APIs, Microsoft has also developed a Gesture API. This API was not included by the W3C in the Pointer Events specification.

The Microsoft API supports detection of pinch, swipe, hold, pan, tap, and long press (tap-and-hold) gestures without reading raw pointer events. To start a gesture recognition mechanism, we must configure an `MSGesture` object when a touch starts (`mspointerdown`) and we want to detect gestures on it.

The Gesture API allows the detection of inertia, which is a way to continue receiving events after the user releases his fingers, following the speed of the original gesture. That is, if the user makes a fast swipe, we can receive inertia events based on the speed of the gesture that allow us to create visual effects that decelerate.



When working with gesture events, it's a good idea to disable zooming, scrolling, and text selection from the elements using `-ms-touch-action: none` and `-ms-user-select: none`.

Starting a gesture detection. To start a gesture detection we need to add an element to the gesture engine. We do this using the `MSGesture()` constructor function, passing the target element to it. For example:

```
var myGesture = new MSGesture();
var myElement = document.getElementById("gesturearea");
myGesture.target = myElement;
```

To detect a gesture in that element, we need to capture `mspointerdown` and pass the touch identification to our gesture using the `addPointer()` method, as in:

```
myElement.addEventListener("mspointerdown", function(event) {
    myGesture.addPointer(event.pointerId);
}, false);
```

When we have the gesture implemented, we can start listening for gesture events in the element. Possible events are listed in Table 15-7.

Table 15-7. Events available in the Microsoft Gesture API

Event	Description	States
<code>msgesturehold</code>	Fired when the user touches down, holds the touch/pointer for a moment, and then releases without moving.	When the user has been holding the pointer down long enough for it to be interpreted as a touch and hold, <code>event.MSGESTURE_FLAG_BEGIN</code> is true; when the user has released the contact without moving the pointer <code>event.MSGESTURE_FLAG_END</code> is true; and when the user has released the contact after moving it, so the gesture was canceled, <code>event.MSGESTURE_FLAG_CANCEL</code> is true.

Event	Description	States
<code>msgesturestart</code>	Fired when a touch has started on the element that may be a gesture.	This event indicates the start of a new contact/gesture.
<code>msgesturechange</code>	Fired after a dynamic gesture change, including pinch, rotate, swipe, and pan/drag.	This event fires every time the user moves one touch, and after the user has removed all the touches but there are still inertia events going on.
<code>msgestureend</code>	Fired after all associated touches/pointers have been removed from the screen and any inertia movement has finished.	This event fires when the user has removed all the touches for this gesture and any inertia processing has stopped.
<code>msgesturetap</code>	Fired after a single, quick interaction with the screen has been made.	This event fires if the touch/pointer is removed quickly, at the same point as where it started.
<code>msinertiastart</code>	Fired after the user has removed all touches but, because of the gesture speed, there is still an inertia that will continue "moving" the pointer.	This event fires when the user has removed all touches and there was enough speed in the gesture for inertia.



We can detect if a gesture is in its inertia phase using the argument's `detail` property, which we compare to `event.MSGESTURE_FLAG_INERTIA`. If we don't want to consider inertia for our effects, we can just read this property.

Gesture change values. When detecting a dynamic gesture we usually use the `msgesturechange` event, which receives an argument with different properties that can be easily matched to CSS transformations, such as `event.translationX` and `event.translationY` for translation from the original point; `event.rotation` for rotation; `event.scale` for scaling; and `event.offsetX` and `event.offsetY` for the center point of the gesture (and the possible transformations).

When detecting swipe (flick) gestures, the `msgesturechange` event has useful properties such as `velocityX` and `velocityY` for the velocity on both axes, and `velocityAngular` for the angular velocity in radians.



Microsoft supports an `mscontentzoom` event that will be fired after the user has zoomed the target element.

To apply a CSS transformation to the element in which we are detecting gestures, we can use the following snippet, based on a Microsoft official documentation example:

```
myElement.addEventListener("MSGestureChange", function(e) {
    // Get the latest CSS transform on the element
```

```

    var m = new MSCSSMatrix(e.target.style.transform);
    e.target.style.transform = m
    // Move the transform origin under the gesture's center
    .translate(e.offsetX, e.offsetY)
    .rotate(e.rotation * 180 / Math.PI) // Apply rotation
    .scale(e.scale) // Apply scale
    .translate(e.translationX, e.translationY) // Apply translation
    .translate(-e.offsetX, -e.offsetY); // Move transform origin back
  }, false);

```

Sensors

Touch is not the only input that we can read from an HTML5 mobile app. Most devices after the iPhone included a couple of sensors that we can also use to provide input data for our apps or games, including the accelerometer, magnetometer (digital compass), and gyroscope. Some devices also include pressure and lighting sensors.

While trying to understand how these sensors work is out of the scope of this book, it's important to know what each sensor is useful for:

Accelerometer

Measures linear motion (current acceleration) on the device on three axes (x , y , z), including gravity. That means that we can detect how the user is holding the device in three dimensions, as well as any gesture that the user may be doing with the device, such as shaking.

Gyroscope

Measures rate of rotation around one axis, based on angular momentum. Gravity does not affect the gyro, and it is usually used in conjunction with the accelerometer to provide trusted information about the device's rotation and orientation.

Magnetometer

Measures the device heading in relation to true north, in degrees. This is the necessary sensor to detect yaw, which is orthogonal to gravity.

Pressure sensor

Measures barometric information.



While you may think that using the accelerometer will consume battery power, it's important to know that this sensor is always on; it's the one behind the screen rotation mechanism.

DeviceOrientation Event

The DeviceOrientation Event W3C (<http://www.w3.org/TR/orientation-event>) specification is a draft that is implemented on most tablet and smartphone platforms today. Safari on iOS 4.2 was the first mobile browser to support it, but today it can be found

in the Android browser (since 3.0), the BlackBerry browser, and many others, as we can see at Mobile HTML5 (<http://www.mobilehtml5.org>).

This specification adds three events that we can attach to the window object: `deviceorientation` for orientation change detection (compatible when a gyro is available), `devicemotion` for accelerometer data detection, and `compassneeds_calibration` for when the device detects that the compass needs a calibration to increase data accuracy.

When using `deviceorientation`, every time the user moves the device we will receive an event call that includes the properties `alpha` (0–360), `beta` (–90–90), and `gamma` (–180–180) for the rotation of the device frame around its *z*-, *x*-, and *y*-axes, respectively. These values are typically defined as an offset from an arbitrary direction, usually the direction in which the device was held when the orientation was first obtained. Therefore, `deviceorientation` is more useful for relative movements from the original position.



When `compassneeds_calibration` is fired we need to inform the user that she needs to make a figure-eight motion in the air to improve accuracy.

The `devicemotion` event shows gravity and user acceleration on devices with both an accelerometer and a gyroscope. iPhone 3GS, iPad first generation, and older iPod touch devices didn't include a gyroscope; therefore, this API can only measure gravity acceleration on those devices.

The event gives us the `rotationRate` (`alpha`, `beta`, and `gamma` values if a gyro is available), `acceleration` (for the user's acceleration if a gyro is available), and `accelerationIncludingGravity` (for user and gravity acceleration). Acceleration properties include `x`, `y`, and `z` for the acceleration in each axis, measured in m/s^2 .



We should always query about ranges of values for the accelerometer and orientation and not exact values. Different ways of measuring can be found between different implementations in different browsers, such as one device giving positive values when facing back while others give negative values for the same orientation. Testing and user calibration are important to provide a multiplatform solution today.

There is no better way to understand accelerometer and gyro values than trying this out yourself on a real device. You can point your mobile browser now to <http://t.ad.ag> and look for the JavaScript API→Device Motion tests.

Moving a ball. One of the classic examples of accelerometer usage is allowing the user to move the device to give acceleration to a virtual table with a little ball on it. To create this particular example using this API, we can use the following working sample:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Move the Ball</title>
<meta name="viewport" content="width=device-width,user-scalable=no">
<style>
#no {
    display: none;
}
#ball {
    width: 20px;
    height: 20px;
    border-radius: 10px;
    background-color: red;
    position: absolute;
    top: 0px;
    left: 0px;
}
</style>
</head>

<body>

<div id="content">
    <h1>Move the Ball</h1>
    <div id="yes">
    <div id="ball"></div>
    </div>
    <div id="no">
        Your browser does not support the Device Orientation
        and Motion API.
    </div>
</div>

<script>
// Position variables
var x = 0;
var y = 0;

// Speed - Velocity
var vx = 0;
var vy = 0;

// Acceleration
var ax = 0;
var ay = 0;
```

```

var delay = 10;
var vMultiplier = 0.01;

if (window.DeviceMotionEvent==undefined) {
    // API not available
    document.getElementById("no").style.display="block";
    document.getElementById("yes").style.display="none";
} else {
    window.addEventListener("devicemotion", function(event) {
        ax = event.accelerationIncludingGravity.x;
        ay = event.accelerationIncludingGravity.y;
    });

    setInterval(function() {
        // We calculate new velocity, based on current acceleration
        vy = vy + -(ay);
        vx = vx + ax;

        // We calculate new position, based on current velocity
        y = parseInt(y + vy * vMultiplier);
        x = parseInt(x + vx * vMultiplier);

        // Detect screen boundaries
        if (x<0) { x = 0; vx = 0; }
        if (y<0) { y = 0; vy = 0; }

        if (x>document.documentElement.clientWidth-20) {
            x = document.documentElement.clientWidth-20; vx = 0;
        }
        if (y>document.documentElement.clientHeight-20) {
            y = document.documentElement.clientHeight-20; vy = 0;
        }

        // "Move" the ball to its new position
        var ball = document.getElementById("ball");
        ball.style.top = y + "px";
        ball.style.left = x + "px";
    }, delay);
}

</script>

</body>
</html>

```

Unless we are creating a native web app, there is no way to force orientation lock in HTML5. Therefore, it's possible that when the user is moving and tilting the device, the orientation will change, and that will lead to some UI problems. We should try to detect these changes and pause/stop the reading until the user goes back to the desired orientation.



Using the accelerometer at the same time as detecting touch events, we can emulate pressure detection, as we can detect how much movement the device had while the user was touching it.

Compass information

According to the specification, to get the compass heading we just need to handle `deviceorientation` and use the formula $360 - \text{event.alpha}$. However, life isn't always that simple, and WebKit-based browsers may differ in their implementation. For example, Apple says that the `deviceorientation` event will show relative information from an arbitrary position and not an absolute position on Earth; therefore, `event.alpha` is not absolute, but a relative measure from the first reading.

On Safari on iOS from 5.0, we can use `event.webkitCompassHeading` to have the heading measured in degrees relative to magnetic north, with 0 being north, 180 south, 90 east, and 270 west. `event.webkitCompassAccuracy` will give us the accuracy measured in degrees; if this property is set to -1 it's because the compass is not calibrated and the information should not be used.

When the user is changing the device's orientation (e.g., landscape to portrait) the heading may differ, so we can use the `window.orientation` value to act and normalize the `webkitCompassHeading` information.

If you want to see a working example of the compass API with CSS transformations, you can check out the open source solution created by James Pearce at his GitHub page (<https://github.com/jamesgpearce/compios5>).

Apache Cordova API

Apache Cordova/PhoneGap includes a proprietary accelerometer and compass API to read raw data when creating this kind of app. Two objects are exposed to access the sensors: `navigator.accelerometer` and `navigator.compass`. The API is similar to the Geolocation API, defining methods such as `getCurrentAcceleration`, `watchAcceleration`, `clearAcceleration`, `getCurrentHeading`, `watchHeading`, and `clearHeading`.

Compass handlers will receive an `event.magneticHeading` attribute and accelerometer handlers will receive `event.x`, `event.y`, and `event.z` values for acceleration on every axis.

Network Information

The Network Information API (<http://www.w3.org/TR/netinfo-api>) allows us to determine some information about the current connection. At the time of this writing, two different implementations of the API are available: an old spec version first implemented by the Android browser and Apache Cordova/PhoneGap apps; and the new W3C spec,

including bandwidth information first implemented by the Blackberry 10 browser and Firefox for mobile devices.

Old specification

The old spec API (<http://bit.ly/10St3bL>) exposes a `navigator.connection.type` attribute that will match one of the following strings: `unknown`, `ethernet`, `wifi`, `2g`, `3g`, `4g`, or `none`.



While the old W3C spec said that the `type` attribute should be a string, the Android browser and Apache Cordova implementations expose its value as an integer, mapped to the `navigator.connection` constants.

This API will give you the connection type of the first node; that doesn't mean that it is working properly or even that it is the ultimate type of connection. For example, it's common to have WiFi on some buses and trains today, and the WiFi access point may be using a 3G connection.

Besides this API, browsers also expose two events—`online` and `offline`—that we can use to detect changes in connection status, as well as the `window.online` Boolean property.



When we are offline, we are sure there is no Internet connection, no WiFi, and no cellular connection. If the API is exposing that we are online, that just means that there is a connection available; it doesn't mean it's working.

New specification

Knowing which type of connection the user has may not be enough—he may be on WiFi under a 2G-connection hotspot or have a nonworking 4G connection. That's why the W3C working group for this API has developed a new draft of it (implemented only in Firefox for Android and the Blackberry 10 browser at the time of this writing).

This new spec is still based on the same `navigator.connection` object, but instead of exposing a `type` attribute it exposes `bandwidth` and `metered` attributes and a `change` event.

The `bandwidth` can be `0` (offline), `infinite` (unknown), or any estimation in MB/s (megabytes per second); `metered` is a Boolean attribute indicating that the current connection may be limited by the Internet provider and that we should be careful about bandwidth usage.

Besides the `online` and `offline` events, this new spec also delivers a new `change` event over `navigator.connection` that will trigger a call every time the connection changes.

Currently Firefox supports this feature through a `moz` prefixed version and Chrome (for desktops only at the time of this writing) with a `webkit` prefixed version.

If this API is available, we can decide what kind of experience we are going to deliver, including SD or HD video and different image qualities, as in:

```
var connection = navigator.connection || navigator.mozConnection ||
    navigator.webkitConnection;

if (connection.bandwidth!=undefined) {
    if (connection.bandwidth <=1) {
        // Less than 1 MB/s => Low quality
    } else if (connection.bandwidth > 1) {
        // More than 1 MB/s => High quality
    } else {
        // Offline or unknown
    }
} else {
    // API not available or older spec
}
```

File Management

The File API (<http://www.w3.org/TR/FileAPI/>) is a W3C specification that is available in some modern mobile browsers, including Google Chrome, the BlackBerry browser, Internet Explorer since version 10, and Safari on iOS since version 6. With this API we can read files from the device's filesystem at the user's request.

On some platforms the same API also allows us to navigate through a filesystem and write files, as in Apache Cordova/PhoneGap apps.



The Directories and System API (also known as the FileSystem API) should not be confused with the File API, as the first one's intent is to create a virtual filesystem for isolated app storage, while the latter works with the user's public filesystem. We can also use the File API on our isolated filesystem.

When working with browser-based apps on mobile devices, this API can be triggered after the user has selected one or more files from a file upload form input control (`<input type="file">`) using the change event. In desktop browsers, this API can also be used with the Drag and Drop API (which is useless on mobile devices as there is no way to drag a file from the filesystem to the browser).



Remember from previous chapters that the file upload control can also be used for taking pictures or recording audio and video using the `accept` and `capture` attributes.

The following example will loop through all the files selected in the form control. Even if we are not using the `multiple` attribute and it's a single selection, we always receive an array of files. Every file object has a name, a size, and a type (the MIME type of the file):

```
<input type="file" id="selectFiles">
<script>
document.getElementById("selectFiles").addEventListener
("change", function(event) {
  for (var i=0; i<event.files.length; i++) {
    var fileObject = event.files[i];
    var fileURL = URL.createObjectURL(fileObject);
    // We are ready to read the file
  }
}, false);
</script>
```

Using the `FileReader` constructor function, we can read files in three different formats: as text, in data URI string format, and as binary data, using the methods `readAsText()`, `readAsDataURL()`, and `readAsBinary()`, with every method receiving the file URL as an argument. The file URL (following the `file://` protocol) can be obtained using the `URL.createObjectURL()` method that is available on the DOM.

File reading is an asynchronous process and we should handle the load and error events of the `FileReader`, receiving in the first event the file data in the requested format:

```
for (var i=0; i<event.files.length; i++) {
  var fileObject = event.files[i];
  var fileURL = URL.createObjectURL(fileObject);
  var fr = new FileReader();
  fr.addEventListener("load", function(e) {
    var data = e.target.result;
    // We can now save, send, or show the contents of the file
    // in data. We can also revoke the URL to free resources
    // and for security purposes.
    URL.revokeObjectURL(fileURL);
  }, false);
  fr.readAsDataURL(fileURL);
}
```

Full FileSystem API

Apache Cordova/PhoneGap apps support the full API, including `FileWriter` to write files, `DirectoryReader` for folder browsing, and `FileTransfer` to upload the files to an HTTP server without `XMLHttpRequest`. For full documentation and working examples, visit Phonegap (<http://docs.phonegap.com>).

Every native app—including native web apps, such as Cordova/PhoneGap apps—usually has two folders available for app storage: a persistent and a temporary storage folder. The persistent folder will keep files for different sessions, and on some platforms that

folder is backed up automatically on the computer or in the cloud (such as in iCloud). To use these filesystems, we need to request one using the `FileSystem` API, as in:

```
window.requestFileSystem(LocalFileSystem.PERSISTENT, 0,
  function(fs) {
    // Filesystem ready to use; we can use DirectoryReader to browse
    // it or FileReader/FileWriter to read and/or write files in it
  },
  function(e) {
    // Failed
  });
```

Full Screen

If we want to get a full-screen experience (no browser UI), we can use the following techniques:

- Package the app as a native web app (see Chapter 16)
- In iOS, create a Home screen web app (see Chapter 16)
- On compatible devices, suggest to the user to force full-screen mode using a browser feature based on current viewport size (at the time of this writing, available only in Safari on iOS 6 in landscape mode and Internet Explorer 10 on Windows 8)
- Use the Full Screen API

The Full Screen API (<http://www.w3.org/TR/fullscreen>) is a way to request the browser to convert one element into a full-screen one, without the browser UI. At the time of this writing, the only browser supporting the API without any bugs is Amazon Silk for the Kindle Fire 2.0. It's possible that other browsers will implement it in the future.

To make a full-screen experience, we need to call the `requestFullscreen()` method of the DOM element we want to force to this mode. Any sibling elements or elements outside of it will not be on the screen while it is in full-screen mode. On WebKit- and Gecko-based browsers, this API is prefixed, so the method is called `webkitRequestFullscreen()` or `mozRequestFullscreen()` instead. It's important to understand that usually the full-screen request can be made only after a user interaction, as a click/tap on the element, and not on load events or timers. For example:

```
function goFullscreen() {
  var element = document.getElementById("fullscreenElement");
  if (element.requestFullscreen) {
    element.requestFullscreen();
  } else if (element.mozRequestFullscreen) {
    element.mozRequestFullscreen();
  } else if (element.webkitRequestFullscreen) {
    element.webkitRequestFullscreen();
  }
}
```

The user usually has a way to get back to the normal web UI through the browser's user interface (in Amazon Silk, there is a UI button), and we can also disable full-screen mode from code by calling `exitFullscreen()` on the document object. The previous version of the specification (available on current browsers) defines the exit operation as `cancelFullScreen()` and all the prefixed versions.



In compatible browsers we can use the new pseudoclass `:fullscreen` to define styles that will apply only in full-screen mode. In other browsers, the pseudoclass must be prefixed, as in `:-webkit-full-screen` or `:-moz-fullscreen`.

The specification (old and new) also adds the Boolean `fullscreenEnabled` attribute and the `fullscreenElement` attribute for getting the current DOM element that is taking up the screen.

Web Notifications

The Web Notifications API (<http://www.w3.org/TR/notifications>) allows a website to add a notification alert into the operating system notification center. This API is not well supported in mobile devices, primarily because only a few platforms allow a background browser operation that can take advantage of notifications to the user.

At the time of this writing, only the BlackBerry browser for PlayBook and BB10 and Firefox support this API, with the `webkit` and `moz` prefixes, respectively. In Figure 15-3 we can see how Firefox delivers a notification on an Android device.



Figure 15-3. Firefox for Android supports the Web Notifications API—after getting the user's approval, we can add messages to the notification bar and receive events from the notification (if the user clicks on it).

The API is based on a global `window.notifications` object that today is prefixed as `window.webkitNotifications` and `window.mozNotifications`. The main problem with this specification is that there are currently two different APIs on the market (the old and new versions).



The old specification of this API was implemented by Chrome for desktops, but at the time of this writing Chrome for Android doesn't support the new API. For compatibility updates, check Mobile HTML5 (<http://www.mobilehtml5.org>).

We need permission from the user for our website to send notifications. To get that permission, we need to call the method `requestPermission()` from the `notifications` object; this method will receive a callback as an argument. The old specification supports a `checkPermission()` function that returns a `0` value if the permission was granted. The current specification has a `permission` property with the string values `default`, `denied`, or `granted`.

Once we have permission, we can generate a notification using the `createNotification()` method in the old spec or the `Notification()` constructor function in the new spec, which receives a title and an object with options.

In both specs we can handle events for the notification, including `display`, `click`, `error`, and `close`.



To further complicate our work, for old spec implementations, Mozilla receives the `icon_url` as the last argument and WebKit-based browsers as the first argument.

In the old spec, to create a notification we can use the following snippet:

```
var notificationManager = navigator.mozNotification ||
    navigator.webkitNotification || navigator.notification;

if (notificationManager!=undefined) {
    var notification;
    if (navigator.mozNotification) {
        notification =
            notificationManager.createNotification("icon_url",
            "title", "message");
    } else {
        notification = notificationManager.createNotification("title",
            "message", "icon_url");
    }
    notification.show();
    notification.onclick = function() {
```

```

    alert('You clicked the notification');
  }
}

```

In the new spec, the same code will look like this:

```

var Notification = window.Notification || window.mozNotification ||
    window.webkitNotification;

if (Notification!==undefined) {
  var options = {
    iconUrl: "icon_url",
    body: "message",
    onclick: function() {
      alert('You clicked the notification');
    }
  }
  var notification = new Notification("title", options);
}

```



In the new spec, in the notification's options we have the ability to define a tag. The tag is a string value that will be used to group different notifications from the same source. For example, if we are alerting the user for each mail she receives, we can group all of them into one notification that is updated as more information is received.

Camera

The ability to access the camera—whether to take a picture, read a QR code, or make a video call—is one of the most interesting features of a mobile web app. Standards around camera access vary, and today there are several different APIs for it:

- HTML Media Capture (covered in Chapter 8), through `<input type="file">`
- The Media Capture and Streams API
- Apache Cordova/PhoneGap's Camera API
- Apache Cordova/PhoneGap's Capture API

getUserMedia

The Media Capture and Streams spec (<http://www.w3.org/TR/mediacapture-streams/>), defines different mechanisms to capture and stream media from the device. The camera part is also known as the `getUserMedia` API and is the most compatible part of the spec today on mobile browsers.

The API supports `MediaStream` objects representing streams of media data, usually audio or video, having an input and an output. To use the camera as an input, we generate a stream object from a `getUserMedia()` call that uses the camera as the input source for the media stream. The stream's output can be a file object or, most probable in a mobile app, a video element on the DOM.



In the future, with the WebRTC (Web Real Time Communication) API, we will be able to use video streams to make video calls or enable two-way video streaming through a web browser.

At the time of this writing, `getUserMedia` is compatible only with Opera Mobile from version 12 and the BlackBerry browser for BB10.



Streaming camera input is the base for augmented reality apps, as we can point to or show elements over or around the camera preview. On mobile web platforms, it's still not possible to create nice augmented reality web experiences.

To get a stream from the device's camera to a video element with the ID `player` we need to use the following code, which we can see in action on a BlackBerry 10 browser in Figure 15-4:

```
window.onload = function(){
    window.URL = window.URL || window.webkitURL;

    navigator.getUserMedia = navigator.getUserMedia ||
                           navigator.webkitGetUserMedia || navigator.mozGetUserMedia;
    var video = document.getElementById("player");

    if (navigator.getUserMedia) {
        // we request a media stream with video and audio
        navigator.getUserMedia({audio:true, video:true},
            function(stream){
                // we create a URL for the input camera stream
                // for the video
                video.src=window.URL.createObjectURL(stream);
                video.play();
            });
    }
}
```

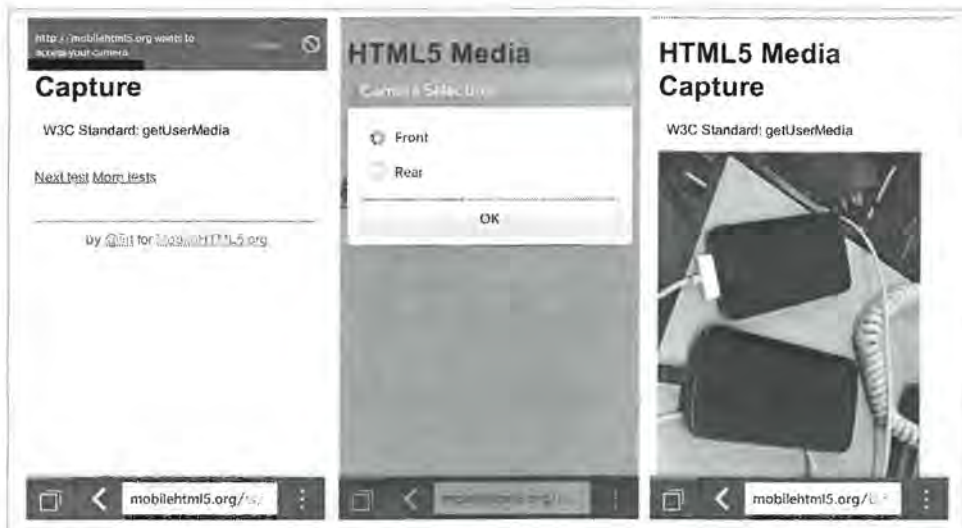


Figure 15-4. In the BlackBerry 10 browser, when we try to open a camera session the user must give permission and select which camera to use before the stream is sent to a video element.



While using a video stream, we can take snapshots of the stream using the Canvas API. To do this, we capture a frame of the video as a source canvas image and convert it to a base64 canvas output image.

Apache Cordova APIs

When using Apache Cordova/PhoneGap, we have two APIs for managing the camera. The simpler API is through `navigator.camera`, such as in:

```
navigator.camera.getPicture(
  function(imageData) {
    // Success callback
    var imageURL = "data:image/jpeg;base64," + imageData;
  }, function(message) {
    // Fail callback
  },
  { // Options
    quality: 50,
    destinationType: Camera.DestinationType.DATA_URL
  }
);
```

We can request the resulting image as a data URI string or as a file URL.

The Capture API is a more complex spec that allows us to record videos, images, or audio from the source. Unfortunately, we will receive the final file and not the stream to show on the screen.

To know more about Cordova APIs, check out <http://docs.phonegap.com>.

Battery

The Battery Status API (<http://www.w3.org/TR/battery-status/>) allows us to read and keep updated on the current battery status, through a `navigator.battery` object. At the time of this writing, only Firefox for Mobile and the Blackberry 10 browser supports it, through a prefixed `navigator.mozBattery` object and a `navigator.webkitBat` tery object.

With this API we can detect if the device is being charged or not through the Boolean `charging` attribute; `chargingTime` is the time remaining in seconds until the battery is fully charged, `dischargingTime` is the time available until a system suspension, and `level` is a float number between 0 (discharged) and 1 (totally charged).

The API also exposes some events through the object, including `chargingchange`, `chargingtimechange`, `dischargingtimechange`, and `levelchange`. The following example shows an alert when the battery is low and not being charged:

```
var battery = navigator.battery || navigator.mozBattery;
battery.levelchange = function() {
  if (battery.level < 0.2) {
    alert('Battery at low level');
  }
}
```

Vibration

The Vibration API (<http://www.w3.org/TR/vibration/>) provides a way to vibrate the phone or tablet, if that feature is available, through the `navigator` global object and a new `vibrate()` function. We can start a vibration, sending one value as the duration expressed in milliseconds, or an array of values. When sending an array of values, the first value will be a vibration duration, the second value will be a pause's duration, and so on. For example:

```
// Vibrate the device 2 seconds
navigator.vibrate(2000);

// Vibrate the device 1s, 0.5s pause, vibrate again 2s
navigator.vibrate([1000, 500, 2000]);
```

To stop any vibration, we need to use `navigator.vibrate(0)`. At the time of this writing, Firefox for Mobile and the Blackberry 10 Browser are the only platforms supporting

this API, through a prefixed `navigator.mozVibrate()` and `navigator.webkitVibrate` function with the same syntax.

Other APIs

There are numerous other APIs that you may want to investigate.

W3C

Other APIs with possibilities in the near future for mobile web applications are:

- Pointer Lock (<http://www.w3.org/TR/pointerlock/>)
- Ambient Light Events (<http://www.w3.org/TR/ambient-light/>)
- Proximity Events (<http://www.w3.org/TR/proximity/>)
- Web Intents (<http://www.w3.org/TR/web-intents/>)
- Pick Contact Intent (<http://www.w3.org/TR/contacts-api/>)
- Messaging (<http://www.w3.org/TR/messaging-api/>)
- Calendar (<http://www.w3.org/TR/calendar-api/>)

The W3C maintains a website following updates on all mobile APIs: the Device API Working Group site (<http://www.w3.org/2009/dap/>).

Apache Cordova

Apache Cordova/PhoneGap apps support other nonstandard APIs, such as:

- Accelerometer
- Contacts
- Globalization

Mozilla

In its WebAPI initiative (<https://wiki.mozilla.org/WebAPI>), Mozilla has a list of APIs that are implemented in Android and/or Firefox OS and are open to other platforms, including some of the APIs we've covered in this chapter, plus:

- WebTelephony
- Payment (for in-app purchases in web apps)
- WebSMS
- Idle
- Power Management

- TCP Socket
- WiFi Information
- Web Bluetooth
- Alarm
- WebFM

For the future

The future of HTML5 and the mobile web includes several APIs that may give more power to HTML5 apps, including:

- WebNFC for Near Field Communication, such as reading RFID tags
- WebRTC for Real Time Communication, including bidirectional audio and video calls
- Web Intents for web app communication
- The Social API for abstract sharing to social networks
- The Session and Identification API for login and authorization purposes



Bowser is an Ericsson Labs browser experiment available for iOS and Android: it's the first mobile browser supporting the full WebRTC draft standard, so you can make video calls between two browsers. More information and demos are available at Ericsson Labs (<https://labs.ericsson.com/apps/bowser>).

Native and Installed Web Apps

Mobile web solutions can run like native installed applications on any platform. This technique is present today in every vendor's roadmap, and many devices are already compatible with some kind of solution for this.

The mobile community hasn't settled on a single name for this kind of application yet; some platforms call them "web apps" and others "offline applications," "JavaScript applications," "hybrid applications," "native web applications," "HTML5 apps," or simply "widgets."

All that said, to simplify our discussion of this kind of application in this chapter, from here on out I will refer to them as *web apps* when they are not packaged and *native web apps* when they are (the power of the author).



In Chapter 3, we discussed the "native versus web" battle and what it means for us.

We'll define a mobile web app as an application entirely developed using web technologies (HTML, CSS, JavaScript) that is installed on the device's Home screen or in the applications menu and that the user can use when offline as well as online. The usage of web technologies is invisible to the user, and the application can work just like any other software installed on the device.

One of the main features of a web app versus hybrid or native web apps is the hosted-based nature: web apps must be hosted on a web server, while native web apps or hybrids are packaged, signed, and distributed through an app store.

Web App Pros and Cons

Web apps are the future for most mobile applications, for a number of reasons:

1. The mobile world is fragmented and will be more fragmented in the future. Chapter 1 covered all the platforms that are available today and likely to be available in the near future. If you want wide coverage you need to create an application that will run on iPhone, Android, Windows Phone, Symbian, BlackBerry 7/10, and Nokia Java ME platforms—and you still won't be covering them all.
2. The web environment demands speed to market: we cannot wait months before releasing our mobile application clients. Web apps can be developed quickly.
3. Every vendor roadmap has a native web platform or similar technology implemented or slated to be implemented in the near future.
4. The majority of a web app's code can be shared between all operating systems.
5. A web app can be a great addition to a mobile website, sharing the same code as the “mobile client” version but offering different possibilities, such as integration with the device.
6. Web apps can be on the users' screens all the time, without requiring them to open the browser and type a URL.
7. Web apps are built using well-known technologies (HTML, JavaScript, CSS) for which a lot of human and technical resources are available.
8. We can use any web API for web app development, without waiting for a mobile SDK to appear.
9. Porting is less painful with web apps than with native applications.
10. It is easy to port mobile web apps from and to desktop widgets (Adobe AIR and others).
11. We can distribute web apps freely or sell them in vendors' stores.
12. Web apps can be self-updated (although on some platforms, such as iOS, store rules don't allow this behavior).
13. With web apps, we can access platform services through new JavaScript APIs not available in mobile web browsers.

However, not everything is golden, and we will face some problems when using this technology:

1. Porting is still required between platforms.
2. Debugging is painful, although this is getting better with time.
3. Web apps are not native applications, so the performance will always be worse than with a native app.

4. Web apps are not suitable for all kinds of applications and games.
5. Web apps are not simple websites, but complete applications using JavaScript; best practices and good programming techniques are mandatory.
6. On most platforms, we cannot create background applications.
7. Nonnative web apps can't access all hardware and operating system services.
8. Have I said yet that there are too many platforms? Certainly more than we want!

Architecture of a Web App

We can define the architecture of a mobile web app as described in Figure 16-1.

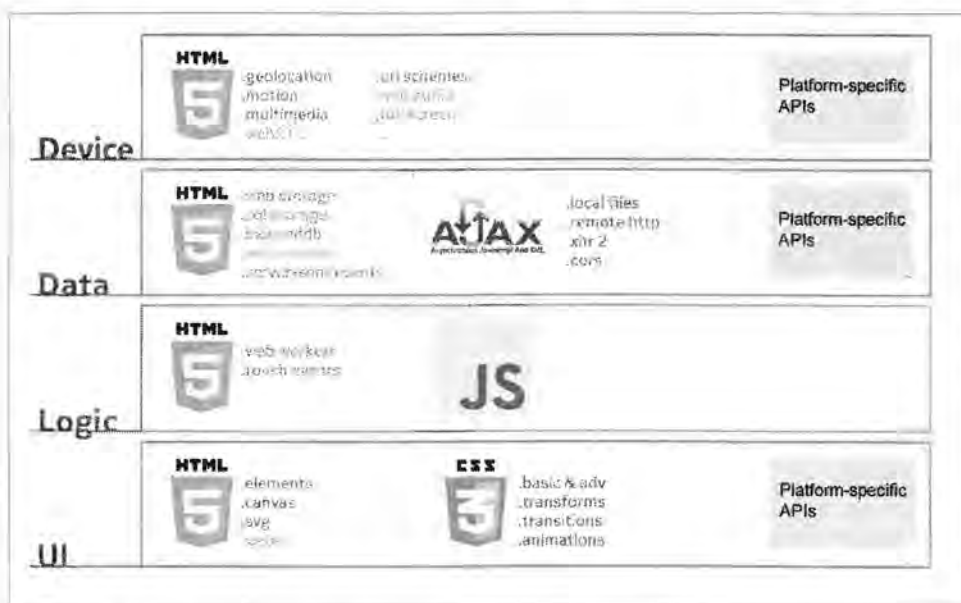


Figure 16-1. Architecture of mobile web app development.

Meta Configuration

Every platform has some kind of metadata configuration file where we generally define the name of the application, the icon to be used for the applications menu, the main HTML or JavaScript file to load when the widget is launched, and other metainformation.

There are widget/web app platforms using all of the following for meta configuration:

- meta tags
- XML files
- JSON files
- Property list (*.plist*) files

Platform Access

Platform access refers to the ability to connect to platform services, using the APIs covered in the previous two chapters.

Data Storage

Web apps are not simple mobile websites; they are applications. And like all applications, they need to store information—databases, configurations, login data, statistics, or whatever else—in some sort of persistent store. As we saw in Chapter 13, HTML5 storage APIs will help us with this.

Network Access

To access the Internet we can use standard Ajax requests, like any JavaScript code, or any other similar solution, such as JSONP requests. Most web app platforms accept cross-platform Ajax requests (to any web server, regardless of the origin of the widget code).



Network communication was also discussed in Chapter 13.

Logic

The entire model, the controller, and the UI logic will be JavaScript code, and using best practices and high-performance object-oriented code will be mandatory. If you want to learn about JavaScript internals, hacks, and how you can write better code, I strongly suggest that you read the excellent book *JavaScript: The Good Parts* by Douglas Crockford (<http://crockford.com>) (O'Reilly), a JavaScript architect at Yahoo!

The first fear about this is, if the source code is plain JavaScript, can't other people look at and even steal our code? The answer is yes, but it shouldn't be a problem. Every Ajax website today (Gmail, Facebook, Hotmail) is JavaScript code that anyone can look at. Also, nothing stops us from using typical obfuscating techniques for our JavaScript code

when we package it up—examining this code will be the same as unpacking a Java ME JAR file or an iPhone native application and trying to decompile the classes. Web apps are no less secure than native applications.

User Interface

The user interface will be defined using all the technologies we've already talked about in this book: HTML, CSS, images, canvas, SVG, and even Flash on supported devices.

Some mobile web app platforms offer us a UI library that we can use to create native-like controls from JavaScript, and some platforms also allow us to define native menus to be used, just like in any other installed application.

Packaging

Every platform offers some kind of package system where we will include all the static assets for our web apps: HTML, JavaScript, CSS, images, text files, configuration files, and any other required resources. Most packages are just ZIP files with a different extension and MIME type. Some platforms can embed a mobile web application inside a native application, and some others will use the HTML5 Application Cache API (the manifest file) to define a virtual package.

Distribution

Finally, when we have our package ready to distribute, we can deliver it to users. Options include over-the-air (OTA) delivery (with the appropriate MIME type applied), providing a URL from which the user can access the application for downloading, or distributing it in stores.

Standards

The standards in this area are still a work in progress, but we can identify some official and de facto standards in the mobile widget world.

Packaging and Configuration Standards

First, for packaging and for the configuration file, the W3C has the Widget Packaging and Configuration standard, defined by W3C (<http://www.w3.org/TR/widgets>) (not only for mobile widgets). The W3C standard defines a ZIP file as the package format, with a configuration file and an optional icon included in the root folder of the package.

The configuration file must be named *config.xml*. Here's a sample file:

```
<?xml version="1.0" encoding="UTF-8"?>
<widget xmlns="http://www.w3.org/ns/widgets"
        id="http://mobilexweb.com/widget">
```

```

<name short="Example 2.0">
    The example Widget!
</name>
<description>
    A sample widget to demonstrate some of the possibilities.
</description>
<icon src="icons/example.png"/>
<content src="myWidget.html"/>
</widget>

```

The other de facto standard is the Apple Dashboard Widget, used for Mac OS X widget development. It also uses a ZIP file, and a property list file (*info.plist*) is used for configuration.

The property list format stores serialized objects in a file with a *.plist* extension. The contents are in XML format, but without the typical XML tag usage.

In a property file, objects are stored along with their properties. Each property can be a string, a number, a Boolean, an array, a key/value dictionary, or some other type, depending on the system. For each property, we define the name as one key tag and the value as another tag, depending on the type. For example:

```

<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
    <dict>
        <key>Numeric Property</key>
        <integer>2010</integer>

        <key>String Property</key>
        <string>Value</string>

        <key>Boolean Property</key>
        <true/>
    </dict>
</plist>

```

Official Platforms

Web apps come in a lot of varieties, as different vendors and organizations have come up with different implementations and technologies over time.

iOS Web Apps

To create JavaScript-based applications for iOS, we have two possible solutions:

- Create a web app (also known as a full-screen web app).
- Create a native web or hybrid solution (for example, a PhoneGap or similar native web project).



A hybrid is a mix between a web and a native application, having the best of both worlds available at the same time.

iOS web app pros and cons

The advantages of a web app are:

- We don't need a Mac-based computer.
- We can host, manage, and change the web app whenever we want.
- We can create any kind of application, including those that Apple doesn't accept as native applications (for example, adult content or private corporate applications for small- and medium-sized companies).
- The application will have an icon in the Home screen.
- The application will be full-screen, and the user will never know it is a web application once it's installed.
- The application can also work offline using application cache.
- We can use all the HTML5 features and APIs we've already seen in this book.



If you create a web app, you can submit it to Apple's web app gallery (<http://www.apple.com/webapps>) for free promotion.

However, there are also some cons:

- We cannot distribute or sell a web app through the App Store (the official Apple store).
- We will not have access to the APIs not exposed in the browser.
- It is not easy to determine whether a web app is already installed on the system.
- Many users still don't know how to install web apps.



With iOS 4.0, Apple created iAd, an advertising program for iOS native applications. The ads are created using HTML5 and some JavaScript extensions. If we want to create these kinds of ads we can use iAd JS (<http://developer.apple.com/iad>), a JavaScript library.

Full-screen meta tag

In iOS 2.1 and later, some new meta tags for full-screen mode are available for web apps. First, we must use the `viewport` meta tag and the `apple-touch-icon` link to provide a 1:1-scale interface and an icon for the Home screen, as covered in Chapter 7:

```
<meta name="viewport" content="width=device-width; initial-scale=1.0;
    maximum-scale=1.0; user-scalable=0;">
<link rel="apple-touch-icon" href="/Icon.png">
```

To hide the entire Safari interface when the application is opened from the Home screen, we use the `apple-mobile-web-app-capable` meta tag with a unique possible value of `yes`:

```
<meta name="apple-mobile-web-app-capable" content="yes">
```

This tag will make no difference if the HTML page is opened in the browser. In Safari on iOS, we can query the `window.navigator.standalone` JavaScript object to see if we are working in standalone mode (`true`) or in browser mode (`false`). The standalone and browser versions of the Sun web app (<http://pattern.dk/sun>) are shown in Figure 16-2.



The latest version of the Nokia Browser for Symbian supports `apple-mobile-web-app-capable` as well as the `mobile-web-app-capable` meta tag for fullscreen standalone mode. First edition of the Blackberry 10 Browser exposes `navigator.standalone` so it may support this mode in later versions.

Changing status bar appearance

We cannot hide the 20-pixel-high top status bar (40 pixels high in iPhone 4 and other high-DPI devices), but we can change its appearance to be compatible with our design. We do this with the `apple-mobile-web-app-status-bar-style` meta tag. This tag allows values of `black`, `default`, or `black-translucent`, and it only works if we have already defined standalone mode.

When we specify the values `black` or `default`, our website will have available the full height of the screen, minus the 20px of the status bar. If we use the `black-translucent` value we will have the entire screen available, and the toolbar will be overlaid on the top 20 pixels of our website.



Figure 16-2. Sun is a simple weather app that detects if the user has accessed it using the browser or the Home screen icon—this is the same HTML file, but the version on the right (the version opened from the Home screen) looks like any other native app.

Defining startup images

From iOS 3.0, Safari also supports a *startup image* to be used as the initial image before the HTML and JavaScript loads:

```
<link rel="apple-touch-startup-image" href="startup.png">
```

There are different image sizes per device type:

- 320×460 pixels for low-resolution iPhone or iPod touch
- 640×920 pixels for high-resolution 3.5" iPhone or iPod touch
- 640×1096 pixels for high-resolution 4" iPhone or iPod touch
- 768×1004 pixels for low-resolution iPad or iPad mini in portrait mode
- 1024×748 pixels for low-resolution iPad or iPad mini in landscape mode

- 1536×2008 pixels for high-resolution iPad in portrait mode
- 2048×1496 pixels for high-resolution iPad in landscape mode



iPad devices have Home screen apps compatible specifically with landscape and portrait modes; that is why we can provide both versions of the launch startup image. iPhones' and iPods' Home screens work only in portrait mode. If we want to invite the user to change the orientation to landscape, we can provide a 90-degree-rotated image.

The application launcher also uses this image for the zoom-in animation when the user clicks on the icon. If you don't supply a startup image, it will use a screenshot from the last time the app was used, or a white screen.



HTML5 Boilerplate includes a JavaScript-based solution that will insert the right startup image for the current device, called `MBP.startupImage`.

If the file size is not correct for the current device, the operating system will just ignore the image. Therefore, we need a way to provide all the possible alternatives. While this solution is undocumented, media queries work on the `link` element for the `apple-touch-startup-image` file. Therefore, we can use this approach to query about the device's screen height, pixel ratio, and/or orientation.



In iOS 6.0 there is a bug that creates a letterboxed app on iPhone 5 and iPod touch 5th generation (4" screen) devices, with black bars at the top and bottom, unless we define a viewport different than 320 or device-width, such as in:

```
<meta name="viewport" content="width=320.1">
```

Fortunately, we can use JavaScript to force that value when we know we are on this kind of device, as in:

```
if (window.screen.height==568) { // iPhone-iPod 4"
  document.querySelector("meta[name=viewport]").
    content="width=320.1";
}
```

All the possible values at the time of this writing are:

```
<link rel="apple-touch-startup-image" href="startup-iPhone-3.5-1x.png"
      media="only screen and (device-height: 480px) and
      (-webkit-device-pixel-ratio: 1)">
<link rel="apple-touch-startup-image" href="startup-iPhone-3.5-2x.png"
      media="only screen and (device-height: 480px) and
      (-webkit-device-pixel-ratio: 2)">
<link rel="apple-touch-startup-image" href="startup-iPhone-4-2x.png"
      media="only screen and (device-height: 568px) and
      (-webkit-device-pixel-ratio: 2)">
<link rel="apple-touch-startup-image"
      href="startup-iPad-1x-landscape.png"
      media="only screen and
      (device-height: 768px) and (-webkit-device-pixel-ratio: 1) and
      (orientation: landscape)">
<link rel="apple-touch-startup-image"
      href="startup-iPad-1x-landscape.png"
      media="only screen and
      (device-height: 768px) and (-webkit-device-pixel-ratio: 1) and
      (orientation: landscape)">
<link rel="apple-touch-startup-image"
      href="startup-iPad-2x-landscape.png"
      media="only screen and
      (device-height: 768px) and (-webkit-device-pixel-ratio: 2) and
      (orientation: landscape)">
```

Unfortunately, at present there is no way to force an orientation, as the user can change it. We can use media queries and/or JavaScript to show a message to the user when she is in an orientation that we don't want to support, however.

How it looks

In the applications menu, a web app looks and acts like any other native app, with the only restriction being that we can't push a badge number update as we can for true native apps.

A web app runs in a different process than Safari (internally called *Web.app*), while still sharing the same rendering engine. If we activate the task list (by double-clicking on the Home screen button), we will see that our web app appears as a completely different process than Safari, as shown in Figure 16-3.

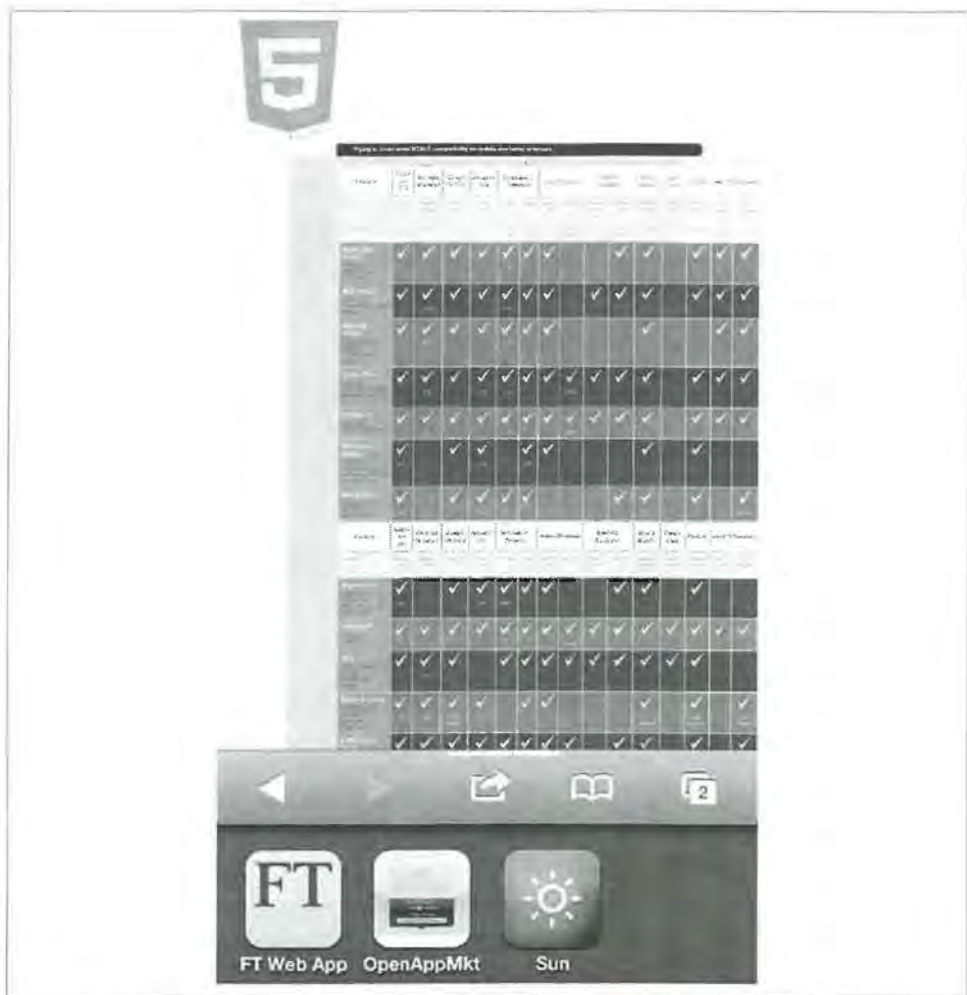


Figure 16-3. In the task list on iOS, every web app appears with its own icon, separate from Safari; in this case, the three icons are full-screen iOS web apps.

The icon is defined by the `apple-touch-icon` link and the name of the app is defined by the `title` tag or the `apple-mobile-web-app-title` meta tag, as we saw in Chapter 7.

APIs

All the same HTML5 APIs are available to us when our website is installed as a web app. Up to iOS 5.1, the same HTML in the browser and as a full-screen web app is considered of the same origin, which means that it shares all the storage data (such as local storage or Web SQL storage).

Starting in iOS 6, web apps are isolated, which means that they have their own storage and the same HTML will access different data when in browser or web app mode. Also, some limits, such as the 50 MB limit for application cache, are lifted when working in full-screen mode.

Links

The iOS web app mechanism expects to serve a single HTML file that will use JavaScript techniques to deliver a rich experience. Because there is no Safari UI in an installed web app, there is no back button or URL bar for the user to use to navigate our app. Therefore, it is our responsibility to provide a full working user interface for all the possible options.

As the web app mechanism is optimized for a single URL, if we make a link to another site (an external link or even a relative URL), that destination will always be opened in Safari, minimizing our app in the operating system.

Using XMLHttpRequest is the preferred way, but if for any reason we want to open a different URL in our current web app context, we can use `location.href` instead, as in:

```
<a href="javascript:location.href='next.html'">Load other file</a>
```

Mobile HTML5 Boilerplate provides a simple way to override the default anchor element behavior:

```
// Safari in standalone mode
if(("standalone" in window.navigator) && window.navigator.standalone){

    // If you want to prevent remote links in standalone web apps
    // from opening in Safari, change 'remotes' to 'true'
    var noddy, remotes = false;

    document.addEventListener('click', function(event) {

        noddy = event.target;

        // Bubble up until we hit link or top HTML element.
        // Warning: BODY element is not compulsory, so better
        // to stop on HTML.
        while(noddy.nodeName !== "A" && noddy.nodeName !== "HTML") {
            noddy = noddy.parentNode;
        }

        if('href' in noddy && noddy.href.indexOf('http') !== -1 &&
            (noddy.href.indexOf(document.location.host) !== -1 ||
            remotes))
        {
            event.preventDefault();
            document.location.href = noddy.href;
        }
    });
}
```

```
    },false);  
}
```

Distribution

As a web app is just a normal web document with some meta tags, the user will not “install” it as such; the “installation” just involves adding the website to the Home screen. However, it is still useful to provide a setup assistant or an installation invitation. The user may not understand the full implementation, so talking about installation is still a good idea in terms of communication.



Remember that some of your users may not be native English speakers. If you are going to give instructions to the user for web app installation, try to provide different language versions, as is done for the device menus.

First, we need to decide if we are going to accept usage of the web app from both the browser and the Home screen, or only as a standalone application. If the last option is our objective, we should provide a single HTML file that detects where the user is accessing it from and either presents the installation link or the app itself (as shown in Figure 16-4), depending on whether it's accessed from the browser or the Home screen.

In the web app HTML file, we first check whether the user is accessing it from the browser or not. If so, we provide instructions for installing the application. For example:

1. Press the + button.
2. Use the “Add to Home Screen” option.
3. The application will be installed on your Home screen for future usage.

The web app HTML should include:

- All the meta tags we've provided
- An offline manifest file (if we want the app to work offline)
- A short title, for use as the application name on the Home screen

We can use any UI library, such as Sencha Touch or jQuery Mobile, to provide an app-like interface.



Figure 16-4. The New York Times web app (<http://app.nytimes.com>) shows a sign-in screen when opened from Safari and not from an installed icon on the Home screen.



Remember that when in standalone mode, the user will not have access to the Back, Forward, or Reload buttons, or to the address bar. Therefore, you should provide all of these navigation items in your design.

Cubiq Add to Home Screen

Add to Home Screen (<http://cubiq.org/add-to-home-screen>) is an open source JavaScript library that creates an invitation bubble for iOS. It includes device detection to offer the right UI for iPad or iPhone/iPod devices, more than 20 localization languages, and other customization features.



Mobile Bookmark Bubble (<http://users.softpress.com/topher/iOSBubble>) is another free library for the “Add to Home Screen” invitation process.

Its basic usage includes code similar to this:

```
<link rel="stylesheet" href="add2home.css">
<script src="add2home.js"></script>
```

The code will show the bubble (as seen in Figure 16-5) only on compatible devices. The framework uses a technique to avoid showing the bubble again in the same session if the user adds the page to the Home screen. Using the advanced setup we can even show the invitation only for returning visitors, to avoid annoying casual visitors who have just landed on our website and are not likely to add it to the Home screen as they don't even know if they like it yet.



Figure 16-5. Invitation bubbles are a must-have feature when working with mobile web apps.



Open App Market (<http://openappmkt.com>) is an open web-based store for HTML5 iOS web apps. Accessing it from Safari on iOS allows us to install the store and then use it to browse through the entire catalog; we can install free web apps and pay for premium web apps.

Symbian Standalone Web Apps

Following Apple syntax, the Symbian browser from version 8.3.1 (mid-2012) supports chromeless full-screen apps installed from the browser. It supports the same `apple-mobile-web-app-capable` attribute (as well as `without-apple-` prefix) and `navigator.standalone` property:

```
<meta name="mobile-web-app-capable" content="yes">
```

If the user adds the page to the Symbian Home screen, the browser will take the current title and it will create an icon (using the techniques described in Chapter 7) that will open the current HTML page in full-screen mode.

Windows 8 Store Apps

Windows 8 supports native store apps developed using HTML5, besides the .NET/XAML development engine. With HTML5 we can create Windows 8 Store apps (previously known as Metro apps) that can be distributed in the Windows Store for desktops, notebooks, and tablets, such as the Microsoft Surface.



As of version 8.0, the official HTML5 engine is not available on Windows Phone; on the Phone version we have to use a web view, such as Apache Cordova/PhoneGap, to compile a native web app with HTML5.

Windows 8 apps feel like full native apps in the user interface and they can use all the same native behavior as .NET apps, as we can see in Figure 16-6 (the Wikipedia HTML5 app, available in the Windows Store).

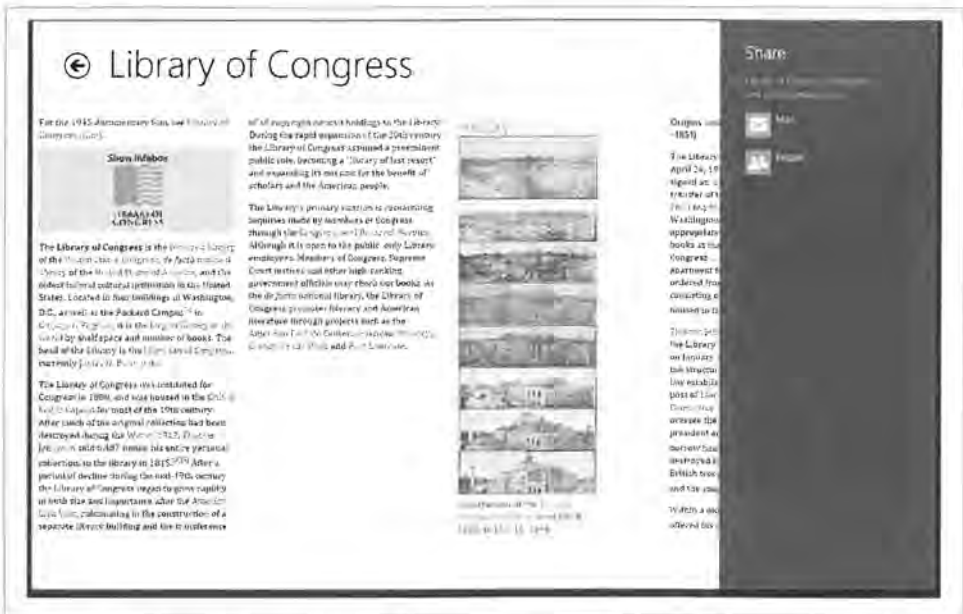


Figure 16-6. The Wikipedia HTML5 app for Windows 8 looks like a native app, including native charm integration such as the Share mechanism.

To develop Windows 8 apps we need:

- A Windows 8 Pro device to develop on
- A free developer license that will enable our Windows 8 Pro device to execute apps that have not yet been approved by the Store

- Visual Studio 2012 or Visual Studio Express 2012 for Windows 8 (free edition)
- Blend for Microsoft Visual Studio 2012
- The Windows 8 SDK

You can download all the tools from Microsoft (<http://go.microsoft.com/fwlink/?LinkId=238221>).



The simplest way to get a developer license is using Visual Studio (even the free Express edition); it will ask if you would like to get one, or you can access the request process from the Store→Acquire Developer License menu option.

To create an HTML5 Windows 8 project, we just use Visual Studio 2012 for Windows 8 and select New Project→JavaScript→Blank App. We can also select one of the many templates already available, as seen in Figure 16-7.



Figure 16-7. With Visual Studio for Windows 8 we can create HTML5 apps for Windows tablets and desktop devices.

While all the code is HTML, CSS, and JavaScript, Windows 8 apps have extensions both in JavaScript and CSS that will make our code look like and behave like a native app.

Windows Runtime

JavaScript apps can access the same Windows Runtime as .NET Store apps, through the global `Windows` object that acts as a namespace. In this object we can find objects and utilities for the core runtime, data and content storage, device management, file management, globalization, graphics, media, networking, printing, presentation, security, and the UI.

There are enough objects, properties, methods, and events available to fill a whole book. Just to give a quick sample, to access the current display's properties, we can use:

```
var displayProperties = Windows.Graphics.Display.DisplayProperties;
```

WinJS

The Windows Library for JavaScript (WinJS) is a package of JavaScript and CSS files that we can import in our HTML using `script` and `link` tags, to provide native features to our app. In the first edition of the platform, all these files are under the folder *//Microsoft.WinJS.1.0/*.

For UI purposes, we will usually add one of the two main CSS theme files—dark or light—using:

```
<link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet">
```

or:

```
<link href="//Microsoft.WinJS.1.0/css/ui-light.css" rel="stylesheet">
```

It's important to use one of these stylesheets, because they provide native style to standard HTML controls, automatic support for different languages, and a high-contrast mode.

In terms of JavaScript libraries, usually we will need at least the base library and the UI library:

```
<script src="//Microsoft.WinJS.1.0/js/base.js"></script>
<script src="//Microsoft.WinJS.1.0/js/ui.js"></script>
```

The base framework will add the `WinJS` global object, where we can find application utilities, namespaces for other objects, and object constructors. One of the main objects is `WinJS.Application`; this points to the application itself and exposes several useful events, properties, and functions. Our app can handle different states through this object, such as suspension and reactivation.

The basic code for handling launch events is:

```
WinJS.Binding.optimizeBindingReferences = true;

var app = WinJS.Application;
var activation = Windows.ApplicationModel.Activation;
```

```

app.onactivated = function (args) {
    if (args.detail.kind === activation.ActivationKind.launch) {
        if (args.detail.previousExecutionState !==
            activation.ApplicationExecutionState.terminated) {
            // TODO: This application has been newly launched.
            // Initialize your application here.
        } else {
            // TODO: This application has been reactivated from
            // suspension. Restore application state here.
        }
        args.setPromise(WinJS.UI.processAll());
    }
};

```

Remember, even if we see native objects in JavaScript, we are still working with HTML, CSS, and JavaScript; therefore, we are still using `addEventListener` or DOM functions for content manipulation.



The `setPromise()` method informs the application that a task is in progress and should not be considered complete until the “promise” completes. We can use this to show a splash screen while initializing objects, for example. The `WinJS.UI.processAll()` function will parse our HTML and search for native object declarations to initialize them.

JavaScript UI controls

While we can create our UI with plain HTML, CSS, SVG, or the Canvas API, we can also use native controls from the operating system—this marks the first big difference between a normal website in Internet Explorer 10 and a native Windows 8 HTML5 app.

These native controls have no markup associated with them, and they will be defined as a `div` element using the `data-win-control` attribute. Because these controls are not associated with a DOM element, their attributes are defined in a JSON format inside a `data-win-options` HTML attribute. For example, to create a Time Picker, we can use:

```

<div id="myTimePicker" data-win-control="WinJS.UI.TimePicker"
    data-win-options="{current: '1:30 pm'}">
</div>

```

To get the control object from JavaScript, we can use any DOM selection mechanism and the `winControl` attribute of the DOM element. For example:

```

var timePicker = document.getElementById("myTimePicker").winControl;

```

Before accessing native controls from JavaScript we need to wait for the application to be initialized, and that process is different from listening for `DOMContentLoaded` or `load` events. Instead, we have to wait for the `processAll` asynchronous process to finish, as shown here:

```
WinJS.UI.processAll().then(function() {
    // Now we can access all native controls here
});
```

The native controls available in WinJS 1.0 are listed in Table 16-1.

Table 16-1. Windows 8 native controls available from HTML5 using `data-win-control="WinJS.UI.<control>"`

Control name	Allows us to...
AppBar	Display commands in the app's context menu (on right-click with mouse or gesture on touch devices).
AppBarCommand	Define a command for the AppBar.
DatePicker	Select a date.
FlipView	Display a collection of items one at a time, such as photos for an album.
Flyout	Display an interactive or noninteractive message in a pop-up that can be dismissed by clicking or tapping outside of it.
HtmlControl	Display an HTML page dynamically.
Menu	Define a menu layout for displaying commands.
PageControl	Define a modular unit of HTML, CSS, and JavaScript that can be navigated to.
Rating	Define a rating control.
SemanticZoom	Define how the user can zoom between two different views supplied by two child controls (zoomed-out and zoomed-in views).
SettingsFlyout	Define in-context access to the application's settings.
TimePicker	Select a time.
ToggleSwitch	Turn an item off or on (Boolean control).
Tooltip	Display rich content to show more information about an object.
ViewBox	Scale a single child element to fill the available space without resizing it.



Every control has a set of attributes, events, and styles that we can use. The official documentation shows all the possibilities at Microsoft (<http://bit.ly/13YnEAT>).

Application UI state

A Windows 8 app can work in different states:

- Full-screen portrait (on tablets)
- Full-screen landscape
- Filled (sharing the available space with another app; our app is taking ~2/3 of the screen)

- Snapped (sharing the available space with another app; our app is taking ~1/3 of the screen)

To provide a different UI layout or design in each state, we can use media queries and the `-ms-view-state` extension (available only in Windows 8 Store apps):

```
@media screen and (-ms-view-state: fullscreen-landscape) {
}

@media screen and (-ms-view-state: filled) {
}

@media screen and (-ms-view-state: snapped) {
}

@media screen and (-ms-view-state: fullscreen-portrait) {
}
```

Navigation

While not mandatory, if we want to have different views and a navigation controller between them we can use the native `PageControl` object mechanism. We can start a project using the Navigation App template or code it on our own.



Remember we are not creating a website, but an app; therefore, there is no browser UI in our app. We need to provide all the navigation ourselves.

When using this navigation mechanism, we will have a navigation controller and different HTML pages (which can have their own CSS and JavaScript code) for each view. Usually, it's a good idea to use a *pages* folder and one subfolder per view with all the resources inside:

```
<div id="contenthost"
  data-win-control="Application.PageControlNavigator"
  data-win-options="{home: '/pages/home/home.html'}">
</div>
```



When using `PageControlNavigator`, we have both the main HTML file (*default.html*) and the current page's HTML (such as *home.html*). It's similar to using an `iframe`, where two HTML files are on the stage at once.

To navigate to a different HTML page we can use the JavaScript `navigate()` method, passing the relative URI:

```
WinJS.Navigation.navigate(URI);
```

We can pass optional state data as a second argument, and we can use a chained `done()` function that will act as an event handler:

```
WinJS.Navigation.navigate("/pages/details/details.html", {id: 444})  
    .done(function() { // OK }, function(e) { // Error });
```



Because some WinJS objects are used frequently, it's common to have global aliases defined, such as `app` for `WinJS.Application` and `nav` for `WinJS.Navigation`.

To provide an automatic back button (as shown in Figure 16-6) that will transition to the previous page in the navigation controller, we can use the following snippet:

```
<button class="win-backbutton" aria-label="Back" disabled></button>
```

Advanced topics

We don't have enough room in this book to cover the entire Windows 8 HTML5 platform. However, you're welcome to continue exploring on your own. Here are some of the things we can use this platform for:

- Creating and managing tiles and secondary tiles
- Toast notifications
- Badges
- Local, push, and periodic notifications
- Saving and restoring state
- Data binding
- Integrating with the Search charm
- Integrating with the Share charm
- Integrating with the Settings charm
- Communicating with other native apps and the operating system
- Uploading and downloading information in the background

Mozilla Open Web Apps

Open Web Apps (<https://developer.mozilla.org/Apps>) (OWA) is a Mozilla project that allows us to create HTML5 apps that will be executed in a web runtime with a different security model than the browser. While Mozilla's intent is to create a multiplatform open environment, at the time of this writing only Firefox for desktops and Android and Firefox OS support this standard.

Firefox has included a web app engine since version 17 for Android and from the first version of Firefox OS. These apps gain an icon in the Home screen menu and they act like a full-screen app without any browser UI. Firefox calls these apps *open web apps*.

In terms of security, Firefox apps can run in different sandboxes, depending on the type of application/content. The different types are defined as follows:

Web content

Normal website in the browser (not an app).

Installed web app

App with a manifest, downloaded and installed through an unauthenticated website or a store that has no quality assurance (no review on quality and security).

Privileged web app

Authenticated application approved by an app store, equivalent in functionality and security to native apps on other mobile platforms.

Certified web app

Reserved for apps approved by carriers or OEMs that can access core APIs, such as apps for replacing the default dialer, changing the system settings, or power management. This category is not intended for third-party apps.

Every API has a different compatibility based on the sandbox where it is being executed.

An open web app uses standard APIs—such as the Application Cache, Web Storage, IDB, and Canvas APIs—unless the feature is not yet available as a standard, in which case Mozilla uses a custom API.

An open web app must be hosted on our own server or repository using a domain on the Internet, even if it's intended for offline usage.

App origin

The platform is optimized so there is only one app per origin (combination of protocol, domain, and port). If we want to host more than one app, we must use subdomains (such as *app1.mydomain.com* and *app2.mydomain.com*) to host the manifest and the resource files. This doesn't mean that we can't use resources from other domains, just that the app must be installed from only one trusted domain.

App manifest

The manifest is a file served with the MIME type `application/x-web-app-manifest+json` that provides the installation metadata. It is a text file in JSON format, usually with the *.webapp* extension. Here's a basic snippet:

```
{
  "name": "My App",
  "description": "The description goes here",
```

```

"launch_path": "/", // Relative URL to the main HTML page of our app
"icons": {
  "128": "/img/icon-128.png" // >=128 is mandatory
},
"developer": {
  "name": "Max Firtman",
  "url": "http://www.mobilexweb.com"
},
"default_locale": "en",
"permissions": {
  "contacts": {
    "description": "Required to read your contact list",
    "access": "read"
  }
}
}
}

```



There is an online manifest validator at App Manifest (<http://appmanifest.org>).

The manifest should include all the API requirements for the user to approve its usage (such as geolocation). The possible properties available for the manifest are listed in Table 16-2.

Table 16-2. Mozilla open web app manifest properties

Property	Description
name	The visible name of your app (<= 128 chars).
description	A description of your app (<= 1,024 chars).
launch_path	An optional launch URL (in the same domain) to be used as the initial HTML page for the app. If not provided, the current URL will be used.
icons	A map of icon sizes and their URLs. Icons must be square. For Firefox OS, 30×30 and 60×60 are recommended; for Firefox for Android and desktop, 48×48, 64×64, 128×128, and 256×256 are recommended.
csp	An optional content security policy for the app.
type	The app's type (web, privileged, or certified).
developer	Optional information about the developer, including the properties name and url.
locales	A map of locale-specific overrides to the data contained in the manifest. Useful to provide names and descriptions in different languages.
default_locale	The default language of the manifest.
installable_from	An array of trusted origins that you allow the installation from (for example, the Mozilla Marketplace).
version	The version number (as a string) for the app.

Property	Description
orientation	A comma-separated list of orientations the app supports. Available options include: portrait, landscape, portrait-primary, landscape-primary, portrait-secondary, landscape-secondary.
permissions	The set of permissions that the app needs. You must list every API you intend to use that requires the user's permission. Permissions have a description so the user knows why you need it and an optional access property defining read, readwrite, readcreate, or createonly. The list of possible permissions is available from Mozilla (https://developer.mozilla.org/docs/Apps/Manifest).
fullscreen	A Boolean attribute that defines whether the app should be opened in full-screen mode.
appcache_path	An optional URL to the application cache manifest file for offline app storage.
activities	Specific to Firefox OS, defines a list of "web activities" that the app listens to. With activities, we can handle certain MIME types or specific activities that other apps will delegate to us.

The full documentation for the web app manifest is available at the Mozilla Developer Network (<https://developer.mozilla.org/docs/Apps/Manifest>).



Because of an Android limitation, Firefox apps don't have an icon in the applications menu; the icon only appears on the Home screen and in the app launcher inside Firefox.

Distribution

We can install a Firefox open web app from any website, or we can distribute it through the Mozilla Marketplace (<https://marketplace.firefox.com/developers>). Mozilla doesn't host the apps for us, so even if we want to distribute an app through the store we need to host the files on our own web server or web repository.

To install a web app from our own website, we can use the Apps API available in Firefox from version 17 through the `navigator.mozApps` object. The basic template to launch an installation process (as seen in Figure 16-8) is:

```
var manifestUrl = "http://mydomain.com/manifest"; // Change this URL
var request = window.navigator.mozApps.install(manifestUrl);
request.onsuccess = function () {
    alert('Installation successful!')
}
request.onerror = function () {
    alert('Installation failed, error: ' + this.error.name);
}
```



Figure 16-8. An open web app being installed from the Marketplace website using the Apps API.

The Apps API offers methods we can use to tell if the app is already installed (`mozApps.getSelf`), get all the apps installed from the current origin (`mozApps.getIn`), and get all the apps installed from any origin (`mozApps.mgmt.getAll`). The latter is known as the apps management API.

Checking if the app is already installed

As an app is always attached to a domain (or subdomain), on any web page hosted on the same domain we can check if the associated app is installed or not:

```
var request = navigator.mozApps.getSelf();
request.onsuccess = function() {
  if (request.result) {
    // we're installed
  } else {
    // not installed
  }
}
```

```
}  
};
```

Chrome Apps

Google Chrome has its own HTML5 apps engine that is available on Windows, Mac OS, Chrome OS, and Linux at the time of this writing. These apps are also known as *packaged apps*, and all the documentation can be found at the Chrome developer site (<http://developer.chrome.com/apps>).

A version of the Chrome HTML5 apps engine for Android has been announced but not yet launched at the time of this writing. While the desktop and notebook platform includes several APIs and a package mechanism, it seems the Android version will be slightly different, at least in how we will need to package the apps.

To keep updated on Chrome apps, please visit the blog (<http://www.mobilexweb.com>) that goes along with this book.

Samsung Web Apps

Samsung has released an SDK and a JavaScript API to create multidevice web apps for Samsung devices, including smartphones and smart TVs. The SDK, documentation, and emulators (<http://developer.samsung.com/SamsungWebAPI>) are available.

BlackBerry WebWorks Apps

BlackBerry launched a new widget engine in 2009 as a first-class citizen of the operating system, starting from Device Software 5.0. The platform was named WebWorks, and it has evolved to support all the BlackBerry platforms, with some kind of portability between them: BlackBerry 5 to 7, BlackBerry PlayBook, and BlackBerry 10.

WebWorks is an open source framework that provides a JavaScript layer over the standard web view available on every BlackBerry platform; therefore, with WebWorks we compile the app as a native app from HTML5 code. Version 5 of the platform is not recommended, as it's too basic and the performance is not good enough to provide even simple list-based apps.

Today, WebWorks is starting to migrate most of its APIs to HTML5 and/or to match the Apache Cordova/PhoneGap framework. Basically all the modern HTML5 APIs can be used in WebWorks, as mentioned in the compatibility tables (<http://mobilehtml5.org>).

Configuration file

The WebWorks configuration file is an XML document following the W3C configuration file is an XML document following the W3C Widget Package specification, with

lots of additions from RIM. The possible properties vary according to the platform we are targeting (5.x to 7.x, PlayBook, or BB10). A basic BB10 file will look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<widget xmlns="http://www.w3.org/ns/widgets"
  xmlns:rim="http://www.blackberry.com/ns/widgets"
  version="2.0.0.0" id="myApplication">

  <author href="http://www.mobilexweb.com.com/"
    rim:copyright="Copyright 2013 Firt">@firt</author>

  <name>Programming the Mobile Web app</name>

  <description>
    A sample application for the book
  </description>

  <rim:permissions>
    <rim:permit>access_shared</rim:permit>
    <rim:permit>access_location_services</rim:permit>
    <rim:permit>use_camera</rim:permit>
  </rim:permissions>

  <!-- Icon for BB10 -->
  <icon src="icons/icon-150.png"/>

  <!-- Landscape and portrait screens for BB10 -->
  <rim:splash src="splash-1280x768.png"/>
  <rim:splash src="splash-768x1280.png"/>

  <!-- Main HTML file -->
  <content src="index.html" rim:allowInvokeParams="true"/>

  <!-- Features and permissions -->
  <feature id="blackberry.ui.dialog"/>

  <feature id="blackberry.app.orientation">
    <param name="mode" value="portrait" />
  </feature>

  <feature id="blackberry.app">
    <param name="backgroundColor" value="0xFFFF0000" />
  </feature>

  <access uri="http://www.mobilexweb.com" subdomains="true">
    <feature id="blackberry.app" required="true" version="1.0.0"/>
    <feature id="blackberry.invoke" required="true" version="1.0.0"/>
  </access>

  <license href="http://www.mobilexweb.com/" />
</widget>
```

The configuration file must be in the root folder of your project, under the name *config.xml*. The possible items we can define are listed in Table 16-3, along with their current compatibility per platform.

Table 16-3. WebWorks configuration file options

Element	Description	Platform availability
access	Specifies that the application can access resources on the Web (the <code>feature</code> element is used as a child element)	All
author	Author information	All
content	Initial HTML page to load	All
rim:background	A page to be used as a background service	Only 5.x to 7.x
description	Description of the app	All
feature	Specifies a feature or functionality that the application can use	All
icon	Icon(s) for the Home screen; PlayBook accepts only one icon, while the other platforms accept multiple icon sizes	All
license	End user and license information	5.x–7.x and BB10
name	The visible name of the application	All
rim:cache	Cache preferences	Only 5.x to 7.x
rim:category	Folder location for the application	Only PlayBook
rim:connection	Connection timeout	Only 5.x to 7.x
rim:invoke-target	Registers your app as the handler for an invocation event	Only BB10
rim:loadingscreen	Properties for the loading screen	5.x–7.x and PlayBook
rim:navigation	Defines cursor navigation behavior	Only 5.x to 7.x
rim:orientation	Defines the screen orientation	5.x–7.x and PlayBook
rim:permissions	Container for list of <code>rim:permit</code> elements specifying app permissions	PlayBook and BB10
rim:splash	Image to display as the splash screen	Only BB10



The configuration file must have an `access` tag for each Internet domain that we are going to contact using Ajax or some other resource request and a `feature` tag for each API that we are going to use.

WebWorks APIs

The WebWorks platform supports the features listed in Table 16-4, if they have previously been defined in the permissions area of the configuration file (in the `feature` tag). These APIs are additions to the standard HTML5 APIs that the platforms may support based on operating system version. The full API list (<https://developer.blackberry.com/html5/apis>) and details on compatibility per platform are available.



These APIs are available on some platforms, including a JS reference to the WebWorks extension APIs. Check the documentation to see where the file is located on every version.

Table 16-4. Most useful JavaScript BlackBerry WebWorks extension APIs

Feature	Object	Allows us to...
Advertising	<code>blackberry.advertising</code>	Show banners on our app (BlackBerry 6.x and 7.x only)
Application	<code>blackberry.app</code>	Access functions and properties for the application, like the background and foreground and Home screen support
BBM	<code>blackberry.bbm</code>	Access the BlackBerry Messenger (BBM) system to share content and connect to the social network
Event	<code>blackberry.event</code>	Access operating system event handling
File I/O	<code>blackberry.io</code>	Access files and directories
Identity	<code>blackberry.identity</code>	Access user identification information (IMEI, PIN, phone number)
Invoke	<code>blackberry.invoke</code>	Interact with other installed applications
Messaging	<code>blackberry.message</code>	Send email
Payments	<code>blackberry.payment</code>	Access the In App Purchase API for charging content/services through the store
Phone	<code>blackberry.phone</code>	Access the Phone application and call logs
PIM	<code>blackberry.pim</code>	Manage the Calendar, Contacts, Tasks, and Memos
Push	<code>blackberry.push</code>	Manage the listener for information pushed from the server
System	<code>blackberry.system</code>	Get and set system information and event listeners
User Interface	<code>blackberry.ui</code>	Manage new JavaScript dialogs and native menus
Utility	<code>blackberry.utils</code>	Access useful utility functions like blob converters or URL parsers

For example, to add an item to the native menu on BlackBerry 7.x smartphones, we should use:

```
var item = new blackberry.ui.menu.MenuItem(false, 1, "Refresh",
    menuHandler);
blackberry.ui.menu.addMenuItem(item);
```



Starting with BB10, WebWorks is migrating its APIs and mechanisms to match the Apache Cordova/PhoneGap project; therefore, for example, we have a `webworksready` event matching the `deviceready` Cordova event.

On BlackBerry 10 devices we can also use platform-specific user interface libraries to implement features such as toast messages, swipe detection, and cover support (the

ability to run code while the app is in the background and to update the cover image and text).

bbUI.js

BlackBerry has released an open source UI framework, similar to jQuery Mobile, to provide a native BlackBerry UI style to WebWorks apps. This framework, called *bbUI.js*, will use different UI concepts if your app is running on BlackBerry 6.x–7.x, PlayBook, or the new BB10, providing the expected UI on each platform.

The project (<https://github.com/blackberry/bbUI.js>) is based on semantic HTML5 code using `data-bb-*` attributes to define roles and behavior. While some controls are only available for BB10, most of the widgets are multiplatform. To make it work, we need to initialize the framework and then push our first screen; every screen is based on an external HTML file.

In Figure 16-9, we can see a sample of what a *bbUI.js* WebWorks application may look like on BB10.



Figure 16-9. A BlackBerry 10 WebWorks application using *bbUI.js*.

The basic template for a WebWorks application is as follows:

```

<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" type="text/css" href="bbui-0.9.4.css">
</link>
  <script type="text/javascript" src="bbui-0.9.4.js">
</script>
  <script type="text/javascript" src="js/webworks-1.0.2.9.js">
</script>
  <script type="text/javascript">
    document.addEventListener('webworksready', function(e) {
      bb.init();
      // Open our first screen
      bb.pushScreen('homeScreen.htm', 'homeScreen');
    }, false);
  </script>
</head>
<body>
</body>
</html>

```

Screens are basically HTML files defining elements with `data-bb-type="screen"`, other options, and HTML or bbUI.js controls as children:

```

<!DOCTYPE html>
<div data-bb-type="screen" data-bb-effect="fade">
  <div data-bb-type="title" data-bb-caption="First Demo"
    data-bb-back-caption="Back"></div>

  <div data-bb-type="image-list" data-bb-images="none"
    data-bb-style="arrowlist">
    <div data-bb-type="item" data-bb-title="A"></div>
    <div data-bb-type="item" data-bb-title="B"></div>
    <div data-bb-type="item" data-bb-title="C"></div>
    <div data-bb-type="item" data-bb-title="D"></div>
    <div data-bb-type="item" data-bb-title="E"></div>
    <div data-bb-type="item" data-bb-title="F"></div>
  </div>

  <h3>You can also use plain HTML here</h3>

  <div data-bb-type="action-bar" data-bb-back-caption="Back"></div>
</div>

```

Packaging and signing

Every platform has its own SDK, including packaging and signing tools that can be downloaded for free from BlackBerry's developer website (<http://developer.blackberry.com/html5>). The available options are:

- WebWorks for BlackBerry 7 or earlier

- WebWorks for BlackBerry PlayBook
- WebWorks for BlackBerry 10

While a few years ago BlackBerry used to offer an Eclipse IDE, today all the SDKs just include command-line tools to package and sign WebWorks applications.

To package a BB10 app, the steps are:

1. Create a folder for your app.
2. Add the *config.xml* configuration file.
3. Add at least one HTML file and optional resources (CSS, JavaScript, images).
4. Find the *webworks-<version>.js* file in your SDK folder, copy it into your package, and add the *script* element to reference it.
5. Create a ZIP archive of your folder with all the contents.
6. Open a terminal session (Command Prompt on Windows).
7. Move to the BlackBerry SDK folder.
8. Compile the application using `bbwp <zip_path> -o <app_path>`.

For the other platforms, the steps are quite similar. The resulting application can be executed only on developer platforms.



Using a command-line tool we can enable the Web Inspector, allowing us to remotely inspect and debug our WebWorks apps.

When we have the *.bar* file (for BB10 and PlayBook), we still need to sign the package before submitting it to the store. The first step is to get a signing key, available for free from BlackBerry's website (<https://www.blackberry.com/SignedKeys>).

To compile and sign the app, we can use the following command line:

```
bbwp <zip_path> -g <KeyStorePassword> -o <app_path>
```



Don't forget the PIN associated with the key, as it's used to validate your key against RIM's server on every signing process. If you forget the PIN, you will need to get a new key.

Ripple

While RIM is not offering any IDE at the time of this writing, there is an official tool called *Ripple* (a Google Chrome plug-in) that you can use to help with packaging and signing and for a basic test of the WebWorks APIs, without the need to use a real device or an emulator.

To download Ripple, you have to download the extension from BlackBerry's developer site (<http://developer.blackberry.com/html5/download>), then open Google Chrome on your desktop, go to Tools→Extensions, and finally drag the Ripple file you have just downloaded from the website onto the Chrome UI.

When you have it installed you will see a cyan icon on Google Chrome that will start the Ripple simulation environment, as seen in Figure 16-10. With the extension enabled, you just need to open the HTML file you want to test. To package and sign, you can use the Build panel before setting all the SDK paths manually (Figure 16-11).



Figure 16-10, *Ripple is a mobile web app simulator that works as a Google Chrome plug-in on your desktop.*



Figure 16-11. Ripple allows you to define SDK and emulator paths, so it will help you in packaging and testing your WebWorks apps.

Distribution

You can distribute a WebWorks app just as you would any other native application, on every platform (in fact, they are the same *.cod* files for 5.x–7.x and *.bar* files for PlayBook and BB10).

To publish applications in App World, you'll need to create an account (<http://appworld.blackberry.com/isvportal>). Registration is free of charge for developers.

Nokia S40 Web Apps

Nokia has created a web app platform for its Series 40 social phone devices. If you remember from the first few chapters, Series 40 devices include feature phones, touch and keypad-based phones, and some cheap full-touch phones that are selling well in emerging markets.

These devices include the Xpress browser—a cloud-based browser from Nokia—and the web app engine works on it. Therefore, our apps will live in the cloud and will be delivered compressed to the user. Figure 16-12 shows examples of web apps running on S40 devices.



Figure 16-12. With S40 web apps, we can create apps for social phones using web technologies and distribute them in emerging markets.

An S40 web app is a ZIP-based package with the `.wgt` extension; for metadata it uses the W3C Widget Interface specification on a `config.xml` file in the root folder of the package. The rest of the package can consist of HTML, CSS, JavaScript, and images files.

The WGT package is always sent to Nokia's cloud servers, where it is compiled into some custom code that is rendered by the Xpress browser engine on the client. While we are testing, we can just open a URL in the simulator or inside a real S40 browser. We can also compile it into its final Java ME form: the WGT package becomes a Java ME application (containing both JAD and JAR files) that on the device itself acts as a shell for an icon and the app's private URL. When the user opens this app from the applications menu, the Xpress browser is launched and the web app is executed in full-screen mode (no browser UI).

Remember, the Xpress browser is a limited engine and our JavaScript will be executed server-side (with the exception of Nokia Mobile Web Library calls, as we'll see shortly).



From version 1.5 of the web app platform, all images inside the wgt package will be cached on the device itself so they are not being downloaded every time the app needs them. Any external image that we link to will be compressed on the server side and delivered per request without any caching storage.

In terms of compatibility, at the time of this writing the platform supports XHTML 1.0, CSS 2.1, and the W3C Geolocation API; it also has basic support of CSS3 transitions for fades or slide effects.

S40 web apps work with the idea of a single HTML file; therefore, if you make a link to another HTML page, it will be opened as a web page with the browser UI. If you want to have different pages in your application, you can just use XMLHttpRequest (Ajax) and inject HTML into the DOM. Remember, all these JavaScript behaviors will be executed in the cloud and the client will just receive a pre-rendered compressed version of the result.



In S40 web apps we should avoid using the *javascript:* protocol in links and use the *onclick* handler instead.

Configuration

The web app configuration is done through a *config.xml* file in the root folder of the WGT package, with the following template:

```
<?xml version="1.0" encoding="UTF-8"?>
<widget xmlns="http://www.w3.org/ns/widgets" height="200"
  id="http://mobilexweb.com/superappid" version="1.0"
  viewmodes="windowed fullscreen" width="200">

  <name short="My webapp">My Super webapp</name>

  <description>
    This is my great super app for Series 40
  </description>

  <author email="myemail@mydomain.com"
    href="http://www.mobilexweb.com">
    Max Firtman
  </author>

  <icon src="icon.png"/>

  <content src="index.html"/>

  <license>
    Copyright forever
  </license>

  <feature name="nokia://s40.nokia.com/SAWRT/2.0" required="true" />

</widget>
```

Mobile Web Library

The Mobile Web Library (MWL) is a built-in JavaScript framework exposed through the `mwL` global object. It includes basic DOM management functions (for hiding/showing elements, applying/deactivating CSS classes, and event handling), and the main advantage over classic JavaScript code or other frameworks is that all the processing is done on the client side. Any other JavaScript call will be executed as normal, with a round-trip to the server, as this is a cloud-based platform.

For the calls to be client-side only, they must come from JavaScript code inside HTML event handlers, as in:

```
<a onclick="mwL.show('#details')">Show Details</a>
```

If on the `onclick` event we call a function and that function calls `mwL.show()`, it will need a round-trip to the server.

Gesture and keypad events

While we can use the `click` event, other mouse events, as well as touch events, are unsupported. However, on touch devices the platform supports a gesture extension called *synthetic events* that works with the unique client-side Mobile Web Library. The same events can be used for managing the keypad events.

All the events can be listened for through `mwL.add<Event-name>Listener`, which receives the target node selector and the code to execute. The last parameter should be a string value, and it will be evaluated as string code (don't use a function handler). If you include only `mwL` calls in the string execution code, you won't need a server round-trip; any other JavaScript call will go to the server.

Possible events are:

- `SwipeLeft`
- `SwipeRight`
- `SwipeUp`
- `SwipeDown`
- `LongPress`
- `NavLeft`
- `NavRight`
- `NavUp`
- `NavDown`

For example, to bind the long-press (or touch-and-hold) event to a JavaScript handler to show a contextual menu, we can use:

```
mw1.addLongPressListener("#myelement", "mw1.show('#menu')");
```

Storage API

While HTML5 storage APIs are not available, a simple storage API from the W3C Widget Interface specification (<http://www.w3.org/TR/widgets-apis>) is supported. With a limit of 500 keys per app and 256 bytes in length per key-value, we can persist some basic data in our app using the `widget.preferences` collection, as in:

```
// Store data
widget.preferences['mykey'] = 'myvalue';
// Read data
var data = widget.preferences['mykey'];
```

Tools

To develop, simulate, package, and test our apps, we can rely on Nokia Web Tools (<http://www.mobilexweb.com/go/nwt>) for Series 40, an Eclipse-based IDE that will do all we need (Figure 16-13). We can download this free tool for Mac, Linux, and Windows.



Figure 16-13. Nokia Web Tools for Series 40 allows us to design, code, simulate, and distribute S40 web apps for Nokia devices.



Dozens of working demos are available at the Nokia developer website that we can use as the base for our own apps.

This IDE has the ability to communicate via Bluetooth with a real S40 device to test our apps directly on the device (always through the proxy). To make this work we need to install the Nokia Xpress browser on the device (if it isn't installed already), as well as the *Series 40 web app Bluetooth Launcher*, available from the S40 website.

If you don't want to spend a lot of time designing or developing your app, you can use the free online tool Xpress Web App Builder (<http://xpresswebapps.nokia.com>). It will create a Series 40 web app from scratch, based on different templates, such as news, video, pictures, and blog content sharing.

You can learn more about S40 web apps at mobilexweb.com/go/s40webapps.

Distribution

Your web apps can be published to the Nokia Store that every new S40 device includes on the Home screen; to do that you should get a publisher account (<http://publish.nokia.com>). The initial fee at the time of this writing is 1 euro. Users can pay for premium content by credit card or, in some countries, via their operators' billing systems. You will receive 70% of the revenue if the user pays with a credit card and 60% of the revenue if the user pays with operator billing.

The Nokia Store accepts WGT files as a possible app file format.

Nokia Symbian Web Apps

The oldest mobile web app platform is Web Runtime (WRT) for Symbian devices. While the Series 40 is a live platform from Nokia, the Symbian web app platform is now obsolete, meaning that Nokia is not evolving it anymore. However, we can still create and package web apps for both older and newer Symbian devices, such as the Nokia 500 or the Nokia 808 Pureview, launched in 2012. Every Symbian device on the market (and there are still lots of them in non-US markets) supports WRT apps.



This platform has been available since 2007; for many years Nokia called its applications *widgets*, but it migrated to the *webapps* name in 2011.

WRT is a first-class citizen mobile web app engine. When a WRT widget is installed, it appears like any other Symbian or Java ME installed application, and might look like Figure 16-14. From the user's perspective, there is no difference between widgets and

native installed applications. And, of course, the widgets are created entirely using web technologies.



Figure 16-14. A WRT widget looks like any other installed application; it even has native menu support created using JavaScript.

Some devices have Home screen support (also known as MiniView), enabling a web app to stay on the device's Home screen all the time, with visual updates possible. This is an excellent solution for applications related to social media, news, or any other information that can benefit from being regularly updated on the user's Home screen.

The platform access JavaScript API does not follow any of the standards we have seen so far, but it has many similarities to Apple's Dashboard Widget API. Modern Symbian devices, from Symbian Anna, also support some HTML5 APIs and CSS3 features in web apps, as listed on Mobile HTML5 (<http://mobilehtml5.org>).



Apache Cordova/PhoneGap has a WRT version that gives us a layer of JavaScript API compatibility with other platforms.

Package

A WRT web app is a ZIP file with a *.wgt* extension, served as *application/x-nokia-widget*. The configuration file follows the Apple Dashboard standard and not the W3C widget package definition, unlike Series 40 web apps; it is a property list (*info.plist*) file with some mandatory information:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Nokia//DTD PLIST 1.0//EN"
    "http://www.nokia.com/NOKIA_COM_1/DTDs/plist-1.0.dtd">
<plist version="1.0">
<dict>
  <key>DisplayName</key>
  <string>Widget Name</string>
  <key>Identifier</key>
  <string>com.mobilexweb.webapp_unique_id</string>
  <key>MainHTML</key>
  <string>Main.html</string>
  <key>Version</key>
  <string>1.0</string>
  <key>AllowNetworkAccess</key>
  <false/>
  <key>MiniViewEnabled</key>
  <true/>
</dict>
</plist>
```

The *DisplayName* is the name that the user will see on the screen, the *Identifier* is an inverse-URL mechanism to identify an application inside the device, *MainHTML* is the first HTML file that will be opened with the application, and the *Version* is used by the Application Manager when the user is downloading the application again. If we define *AllowNetworkAccess* as *false*, we will not have any access to the Web (including Ajax requests or resource loading). *MiniViewEnabled* is for compatible devices only.

The icon must be in PNG format (the recommended size for the best compatibility is 88×88 pixels), must be named *icon.png*, and must be located in the root folder of the package.

Features

WRT creates three new global objects in the JavaScript context: *widget*, *device*, and *menu*.



WRT allows us to define multilanguage applications; the version corresponding to the user's defined language will automatically be selected.

In WRT 1.0 we also have a `sysinfo` object that allows us to access the System Information API, which we can use to access properties relating to the battery, network information, lights, vibration, beep tone, and memory, as well as filesystem information and system language information services. WRT 1.0 doesn't have access to other APIs.

The device object allows us to use the Platform Services API library in WRT 1.1. The standard version of Platform Services is 1.0; if we are using Platform Services 2.0, we can use some new APIs. Table 16-5 lists the available APIs.

Table 16-5. Nokia Platform Services APIs

API	Allows us to...
AppManager	List applications and launch a specific application or the default handler for a document type.
Calendar	Create, access, and manage Calendar entries.
Contact	Create, access, and manage Contact entries. In Platform Services 2.0, we can also access Contact Groups.
Landmarks	Create, access, and manage Landmark entries that are used by many map applications inside the device. Platform Services 2.0 also allows us to access the Landmarks local database.
Location	Retrieve information about the user's location.
Logging	Retrieve information about call, messaging, and data logs.
Media Management	Retrieve information about media files stored on the device.
Messaging	Send and receive messages.
Sensors	Access physical sensors on the device (like the accelerometer).
System Information	Retrieve system information (similar to WRT 1.0's System Information API).
Camera (2.0)	Launch the Camera application and retrieve information on pictures taken (Platform Services 2.0 only).

If you are targeting devices with WRT 1.x, you can add support for Platform Services 2.0. To do that, download the API and include a *platformservices.js* file in your package and code. You will then be able to access any API using the shortcut `nokia.device.load(interface_name)`.



You can create hybrid applications for Symbian and MeeGo devices using the QtWebKit project. You can find information about porting WRT widgets to QtWebKit at Nokia's developer site (<http://bit.ly/ZqSJHF>).

JavaScript API

The widget object has the methods and properties listed in Table 16-6.

Table 16-6. Methods and properties of the native widget object

Method/property	Description
<code>openURL(url)</code>	Opens the specified URL in a browser window, leaving our widget in the background.
<code>setPreferenceForKey(value, key)</code>	Stores a persistent object (the <i>value</i>) for a specific <i>key</i> that can be read by the same widget anytime. Note that the parameter order is <i>value</i> , <i>key</i> .
<code>preferenceForKey(key)</code>	Retrieves a stored preference for a key, or returns <code>undefined</code> if the key doesn't exist.
<code>prepareForTransition("fade")</code>	Blocks any update on the UI until <code>performTransition()</code> is invoked. This is useful if we are going to change the UI for some controls and we don't want a flick effect.
<code>performTransition()</code>	Updates the UI with the changes made since the <code>prepareForTransition()</code> call.
<code>setNavigationEnabled(Boolean)</code>	Toggles the navigation mode between the default (cursor-based with a pointer on the screen) and focus (tabbed) navigation.
<code>setNavigationType(mode)</code>	Changes the navigation mode (you can select <code>cursor</code> , <code>tabbed</code> , or <code>none</code>). If <code>none</code> is selected, all the key events can be handled by our code. Available since WRT 7.1.
<code>openApplication(uid, param)</code>	Launches an S60 application, identified by its hexadecimal number. There is a list of common UIDs in the documentation.
<code>setDisplayLandscape()</code>	Changes the UI to the landscape orientation.
<code>setDisplayPortrait()</code>	Changes the UI to the portrait orientation.
<code>onshow</code>	Fired when the application comes to the foreground.
<code>onhide</code>	Fired when the application goes to the background.
<code>onexit</code>	Fired when the user presses Exit.
<code>isrotationsupported</code>	Boolean indicating whether we can change the orientation on this device.

With the menu object and the MenuItem class, we can create native menus, and we can define the label and handler for the left soft key with `menu.setLeftSoftkeyLabel(label, handler)`. The right soft key is by default handled by the platform with an "Exit" label, but after WRT 7.1 we can override it using `menu.setRightSoftkeyLabel(label, handler)`. We can also hide and show the soft key labels using `showSoftkeys()` and `hideSoftkeys()`.

The `onShow` event of the menu object will fire when the user opens the menu.

If we don't define a left soft key, by default it will be an "Options" submenu displaying the native menu we created. A MenuItem can have a label, an id for finding the element, an `onSelect` event, and optionally child MenuItem objects for submenus. For example:

```
// We define a label and a menu ID
var option1 = new MenuItem("Refresh", 2);
var option2 = new MenuItem("New item", 3);

// We can use the same handler and use the ID to tell
// which one was pressed
option1.onSelect = menuSelected;
option2.onSelect = menuSelected;
```

```
// We append the first option
menu.append(option1);

// We create a third option with a submenu
menu.append(new MenuItem("Submenu", 4));
// We can search for a MenuItem using the ID
menu.getMenuItemById(4).append(option2);

function menuSelected(id) {
    switch (id) {
        // We can query the ID to decide what to do
    }
}
```



After WRT 1.1, the widget object has a `wrt` property that we can query to get information about the current device and platform, including `widget.wrt.version`, `widget.wrt.platform.model`, and `widget.wrt.platform.comVersion`.

MiniView

The MiniView, or Home screen widget, allows us to display a widget's content (continuously updated) on the device's Home screen, as shown in Figure 16-15. On compatible devices, the widget is installed as normal, but if the `MiniViewEnabled` property is defined as `true` in the `info.plist` file the user can opt to add it to the Home screen.



Figure 16-15. With the MiniView, users can add our mobile web content to their Home screens; using JavaScript, the information displayed in the widget can be updated on a regular basis.

When a widget is displayed on the Home screen, it shows the same HTML file it would if it were being viewed as a full-screen application. It is up to us to detect the window size change and maybe show and hide a `div` depending on the situation. At the time of this writing, the MiniView size is 312×82 pixels, so we can use a `div` with those proportions when we detect that our widget is being displayed on the Home screen. When the user clicks on the widget in the Home screen it will change to a full-screen display, and when the user exits the widget it will again become small (without actually exiting). The script will be running all the time, so to save the device's battery we should keep our background code to the minimum.



When the user adds a widget to the Home screen, he will receive a confirmation dialog that will allow the web app to make any API call at any time in the future without new confirmation dialogs. So, we can safely use any API (with care, please) in the background.

To update the Home screen UI, we can have a timer defined in JavaScript that queries a server via Ajax, or we can use any other API available on the device (like Location).

We can use the `onshow`, `onhide`, and `onresize` events to detect whether the widget is in full-screen or MiniView mode.



Some devices, like the Nokia N8, allow multipage Home screen support, so the user has more space to add widgets to the Home screen.

Tools and libraries

We can use any tools we like to create WRT widgets, starting with any text editor and a ZIP packager. However, Nokia has an Eclipse-based tool that will help us in developing, testing, debugging, and packaging Symbian web apps known as Nokia Web Tools 1.2 (<http://www.mobilexweb.com/go/nwt12>) for Symbian.

Distribution

A WRT web app can be distributed in many ways, including OTA installation from your own server, offline installation from a desktop, and distribution for free or as a premium application in the Nokia Store.

The requirements to publish WRT apps in the Nokia Store are the same as for Series 40 web apps.

Apache Cordova/PhoneGap Apps

In Chapter 3, we talked about web view–based development and introduced PhoneGap and Apache Cordova (recall that Apache Cordova is the open source engine and PhoneGap is the Adobe implementation). Now it's time to start talking about how to use Cordova for creating native web applications.



Remember that a Cordova app is a native app from a user's perspective, in terms of installation and usage. It's up to our UI if it “feels” native or webby. If it feels too webby on iOS, Apple may reject it.

Apache Cordova is compatible with the following platforms at the time of this writing:

- iOS
- Android
- Windows Phone
- Windows 8
- BlackBerry
- Bada
- Symbian (deprecated, with a fully working version up to Cordova 1.0)

There are currently also implementations being developed for other platforms, such as MeeGo, Ubuntu for Phones, and Tizen.

The apps that we create are native apps with a full-screen enhanced web view. From a store's perspective it's just a native app compiled for the target platform.



Apple doesn't allow “packaged websites” as valid apps in the App Store; therefore, if your app looks like a website (with links instead of buttons, for example) it may be rejected even if technically it works properly.

On every platform, Cordova uses the web view that the operating system exposes; that means our code may execute and render differently on every platform, as Cordova is not adding any rendering engine.

PhoneGap Build

Adobe has launched a service that you can use to compile Apache Cordova apps in the cloud without the need to have every SDK installed on your computer, and even without the need to have a Mac for iOS app compilation.

PhoneGap Build has a free account option that will let you create one private app and infinite public apps; you can also sign up for a premium account that allows you to create up to 25 private apps. To create an account, visit PhoneGap Build's website (<http://build.phonegap.com>); in just a few seconds you will be ready to compile Cordova apps for multiple platforms.

PhoneGap Build is integrated into the Adobe Creative Cloud services, so you can sign up for it as a standalone service or simply use your Creative Cloud membership, if you already have one. You can set up an app from a Git server (including GitHub) where you can push updates, or you can upload a ZIP file with the content of your web app.

The ZIP file should include the main HTML file and every resource—CSS, JavaScript, images, data—that the Cordova app is using. We don't need to provide the CordovaJS file as it varies per platform; the cloud service will inject the right JS file for each compilation, we just need to reference a *cordova.js* generic file in our HTML.

To change the default values, including permissions that we request from the operating system, icons, the splash screen, as well as other platform-specific metadata, PhoneGap Build uses a *config.xml* file in the root of the project, following the W3C Widget Packaging specification. You'll find the documentation for this configuration file at PhoneGap Build's documentation page (<http://build.phonegap.com/docs>). PhoneGap Build is a commercial service from Adobe and we should not confuse it with Apache Cordova's openness.

What Cordova does is to enhance all the web views with a common set of APIs that bridge from JavaScript to native code; it also adds several hacks and automatic configuration so the web view will work as expected.

Even though Cordova uses the web view available on each platform, the behavior of some APIs and features may not be the same as in the browser; therefore, testing is crucial for the project's success.



Apache Cordova/PhoneGap doesn't compile apps; it is just a framework that will help us to compile apps using the official SDK on each individual platform. Don't confuse this with PhoneGap Build, a commercial—with a free option—service from Adobe that will compile Cordova apps in the cloud for us.

Template Creation

When you download the latest Apache Cordova package from Apache Incubator (<http://incubator.apache.org/cordova>) or PhoneGap (<http://phonegap.com>), you will receive a ZIP file with one folder per official platform under the *lib* root folder.



Every compilation has its own “Getting Started Guide” as well as its own permissions and Cordova settings files. You’ll find all the guides and documentation at docs.phonegap.com.

At the time of this writing, iOS, Android, and BlackBerry have shell commands to create, build, and emulate Cordova projects from the command line. On the other platforms, we need to do this on our own.

In the `<platform>/bin` folder, such as `android/bin`, we will find a *create* shell command (for Linux and Mac) and a *create.bat* command for Windows. To create a file we need to execute this shell command (remember to add *.bat* on Windows), using the following syntax:

```
create path_to_new_project package_name project_name
```

The *path* should be any valid path on the current operating system; the *package_name* is a unique identifier using the reverse domain name convention (such as `com.oreilly.progmobleweb`), and the project name will be the name of the Xcode, Eclipse, or BlackBerry SDK-created project. For example, on Windows we can do:

```
create.bat c:\apps\mybook com.mobilexweb.book MyBook
```

Remember to navigate first to the folder containing the *create* shell command, add it to the default paths, or use a full-path call.



On BlackBerry, the *package_name* attribute must be omitted.

When the project is created on all three platforms, we will find a *www* or similar folder where all our HTML files and resources should be located, and a *cordova* folder full of tools to use. In that folder we will find the shell commands *debug[.bat]* for building and *emulate[.bat]* for opening the app. For a full list of commands supported, check the documentation.

Cordova Web View

If we are creating a native app for iOS or Android and we just want to use Cordova in one specific part of it instead of creating a full Cordova app, we can use a feature known

as Cordova Web View. On both platforms, we will have a component that inherits the default web view, adding all the Cordova functionality ready to use in our apps.

CordovaJS

As we said, Cordova adds a layer of APIs that are not available by default on most HTML5 platforms today. This API is now known as *CordovaJS*, and it includes a JavaScript file (different per platform) that acts as a bridge to a native library on each platform.

We will find the right file for the platform after creating a project with the command-line tools or inside the platform folder, usually under the name *cordova.js* or *cordova-<version>.js*.



At the time of this writing, even though the CordovaJS public interface may be the same for all platforms, the file you need to include varies per platform. Therefore, if we use the iOS version in an Android project, it will compile but the whole JavaScript API will not work.

DeviceReady event

One of the first things we need to understand about CordovaJS is that most of its functionality relies on a native library that Cordova includes for every platform. Loading that native library can take some milliseconds, and we should wait until it's done before trying to use a Cordova API.

To solve this problem, Cordova adds a new event to the DOM document element: *deviceready*. Even some HTML5 APIs should be executed only after *deviceready*, as Cordova is adding a compatibility layer or changing the default web engine behavior. One good example is the Geolocation API on some platforms, such as iOS: if we don't include the CordovaJS file in our project or if we don't wait for *deviceready*, the geolocation will work, but with an "index.html wants to get your location" message. Showing "index.html" in an app doesn't look so good; that's why CordovaJS overrides the default Geolocation API with a native implementation (using the same syntax) that will look more natural to the user.

To detect the event, we use the standard event-handling mechanism:

```
document.addEventListener("deviceready", function() {  
    // Cordova is ready  
}, false);
```

Other events

Different events are available on the document element after `deviceready` has fired. They may not all work on every platform (check the documentation for details), but the list of most useful events includes:

`backbutton`

Fired when the Back button is pressed on Windows Phone, BlackBerry, or Android devices.

`menubutton`

Fired when the Menu button is pressed on Android or BlackBerry devices.

`searchbutton`

Fired when the Search button is pressed on select Android devices.

`pause`

Fired when our app is put into the background; we can run only quick processes here before the app is frozen on some platforms.

`resume`

Fired when the app is brought back to the foreground. On some platforms our app may be reloaded when it comes back from the background.

`batterystatus`

Fired after the battery status has changed.

`batterylow`

Fired when the device has reached a low battery threshold.

`batterycritical`

Fired when the device has reached a critical battery threshold (and may be close to shutdown).



By default, a Cordova app may be offline in terms of accessing web resources, such as for image loading, accessing analytics scripts, or using XHR to load content. To allow Cordova to use the Internet, we need to add permission. The process varies per platform: we need to edit *res/xml/cordova.xml* on Android, *www/config.xml* on BlackBerry, or *Cordova.plist* on iOS. We can add permission to access the whole Web using the star (*), or whitelist certain domains.

APIs

Some API usage will require permission in the platform's configuration file; that information will be found in the API documentation available in the *doc* folder of the Cordova package, or at docs.phonegap.com.

Cordova also adds a compatibility layer on some platforms for certain HTML5 APIs, such as Geolocation on Symbian, Web SQL on Windows Phone, or a local storage native implementation. Therefore, these APIs will work on those platforms even if the native engine behind doesn't support them.

In addition to the default HTML5 compatibility, CordovaJS adds to most every platform:

- The W3C Network Information API
- The W3C Contacts API, through the `navigator.contacts` object
- The W3C File API
- The Cordova Accelerometer API, through the `navigator.accelerometer` object
- The Cordova Camera API, through the `navigator.camera` object
- The Cordova Capture API
- The Cordova Compass API, through the `navigator.compass` object
- The Cordova Device API, through the `window.device` object
- The Cordova Media API, through the `window.media` object
- The Cordova Notification API, through the `navigator.notification` object



On iOS from version 6 we can debug Cordova apps (in fact, any web view) using Safari's Remote Web Inspector, including JavaScript debugging and profiling.

Notification API. While we have already covered some of these APIs in previous chapters, it is worth taking a quick look at the Notification API at this point.



The Cordova Notification API has nothing to do with push notification services on mobile devices.

The Notification API allows us to use native dialogs instead of web dialogs, such as `alert` and `confirm`. The default dialogs will work, but on some platforms they will not offer the best experience (for example, showing "index.html" as the dialog title). Therefore, CordovaJS adds `navigator.notification.alert()` and `navigator.notification.confirm()` methods that allow us to customize the title, message, and buttons in these dialogs on some platforms.

The following example shows usage of both:

```

navigator.notification.alert("Message",
    function() {
        /* callback */
    }, "Title", "Done button name");

navigator.notification.confirm("Message",
    function(buttonIndex) {
        /* callback */
    }, "Title", "Button1,Button2");

```

Unfortunately, there is no Cordova implementation for the prompt dialog. The Notification API also exposes a `vibrate()` and a `beep()` method.

Plug-ins

If you can't find what you need using an HTML5 API or a CordovaJS API, don't give up—you may be able to use a Cordova plug-in. A plug-in is a mix of a JavaScript public interface and a native library that can access any native API or feature on every platform. The main problem of plug-ins is that for multiplatform compatibility we need to offer different native code (Objective-C for iOS, Java for Android, Java for BlackBerry 6/7, AIR for PlayBook, .NET for Windows Phone, etc.).

A list of currently available open source plug-ins is located at PhoneGap's GitHub (<https://github.com/phonegap/phonegap-plugins>).

Distribution

To distribute a native web application, we need to join every publisher's application store program and follow the same instructions as for full native apps.

iOS App Store

For iOS, you need to join the iOS Developer Program (<http://developer.apple.com/ios/program>). A standard account (personal or business) costs \$99 per year. Without an account, your application will only work on the Simulator.

Once your membership has been approved you will have the ability to test your application on real devices, or you can define up to 100 beta testers.

If you apply for a standard account, you will also have the ability to digitally sign your application to be published to the App Store, as a free or premium application. For premium applications, you will receive 70% of the revenue received.



Wikipedia has decided to use Cordova as the platform for its native apps. At the time of this writing, iOS, Android, Windows Phone, Windows 8, and BlackBerry versions are available, and the whole project is open source; therefore, if you want to take a look at Wikipedia Cordova source code you can just point to <https://github.com/wikimedia/WikipediaMobile>.

Android Stores

You can apply for a Google Play Store publisher account (<http://play.google.com/apps/publish>); the initial fee is \$25. You will receive 70% of the revenue from your premium web applications.

You can also distribute your Android applications via the Amazon Appstore. To sign up for a free publisher account, go to Amazon's app developer page (<http://developer.amazon.com/apps/apps>).

BlackBerry AppWorld

Publishing to the AppWorld is free of charge, and you can register for a publisher account at the Vendor Portal for BlackBerry World (<https://appworld.blackberry.com/ispportal>). To sign your BlackBerry apps (required before uploading files to the store) you need to get a free signature from BlackBerry (<http://www.blackberry.com/SignedKeys>). BlackBerry's store supports apps for smartphones, PlayBooks, and the new BB10, and it follows a similar revenue sharing model as the other stores.

Microsoft Windows Store

For Windows 8, you can get a Store publisher account at the Microsoft Dev Center (<http://msdn.microsoft.com/windows/apps>) (the price may vary based on the developer's country) This enables you to publish free and premium applications for Intel/AMD classic Windows devices and Windows RT (ARM-based) devices such as the Microsoft Surface.

Microsoft Windows Phone Store

Windows Phone has its own store, previously known as Microsoft Marketplace. To create a publisher account you need to go to dev.windowsphone.com and pay the \$99 annual fee. This allows you to publish free or premium apps for every Windows Phone manufacturer, including Nokia.



We covered how to sign up to the Nokia Store and distribute Symbian or S40 web apps earlier in this chapter.

Full-Screen App Patterns

Web app development requires new design patterns to solve the challenges presented. We need to think of our apps as applications, not websites, so some of the techniques we've used for websites will not be useful here.

In the previous chapters we have seen how to technically implement most of the patterns we are going to discuss in this section. We will not talk about pages here, but rather screens or views.

Multiple Views

The first problem stems from the lack of a browser's toolbar for navigation. Typical web links are bad practice because the user will see a flicker effect, and the back feature needs to be implemented by us.

That is why in a web app, we will generally have only one HTML file and then, using JavaScript, will change the view, using static hidden divs, dynamic content generated by code, Ajax replacements, or other similar solutions.

We will generally use one of the following multiview mechanisms:

- Tab navigation (top or bottom)
- A top toolbar for going back and/or a back key handler, if available
- Key and touch paginating for sequential views (like a slideshow)

Layout

Even if we are working with only one web app platform, we need to support different screen sizes, orientation modes, and physical screen dimensions. We'll need to decide whether to use a fixed or a liquid design, or a responsive web design approach.

Input Method

The same problem we encountered when designing mobile websites appears here: the devices on which our web apps are used can have a variety of input methods (touch, keyboards, touchpads, etc.), and we will need to handle all of this by code, supporting all the possible input methods and perhaps changing the layout on touch devices (for larger components).

One-View Widget

A one-view widget is the simplest (and sometimes the most powerful) kind of web app that we can create. It generally has an information view and, optionally, a second view for configuring the details for the main view. This is suitable for weather, financial, social networking, news, and corporate indicators.

Dynamic Application Engine

As JavaScript is a dynamic language, we can easily execute code received by a server (using `eval()` or a JSONP request). With this in mind, we can easily create a web app that is only a little engine (like a web browser) that will receive instructions from our server.

All the code (including access to private APIs on some platforms) can be delivered from the server and optionally cached locally in persistent storage. Thus, we can create a self-updating widget mechanism so that no intervention is required from the user to receive the latest version of the application.

There are some security risks and problems, but for most platforms we can easily create a self-update mechanism.

Reload My Web App

Following on from the previous pattern, while we are developing our apps, adhering to a classic development pattern is not a good idea. That is, we should avoid the process of coding, packaging, signing, and installing every time we want to make a change and see it take effect on a real device or an emulator.

To really accelerate development and testing, we may run our own web server (on the Internet or on the local network) and create a very basic web app that will just load the rest of the files (including scripts, styles, and resources) from our repository. Besides that, we add a simple JavaScript reload feature (`history.reload()`) that we remove when deploying the final version. We then package, sign, and install that application, and every time we want to make a change all we have to do is reload it on the device: all the files will be reloaded from the repository.

When we are done with the testing, we can then move all the files to the package, and create the final web app.

Live Reload

An extension to the previous tip is to create a live reload mechanism. This pattern involves JavaScript code that we add while in debug mode that will open a session to a server, usually with WebSockets or some other network API. On our development machine we have a resident application (or it can be our own IDE) that detects changes to

the filesystem and triggers an update to the web app. Therefore, any time we make a change to a CSS, HTML, JavaScript, or image file, the web app on our testing device will update automatically, without even reloading in some cases.



Live Reload (<http://livereload.com>) offers apps for Mac and Windows that will automatically refresh a page using WebSockets when it finds changes on the filesystem.

CHAPTER 17

Content Delivery

Content delivery is a common need in the mobile web world. Applications, music, video, wallpapers, and any other content can be delivered to compatible devices, but this requires a bit of explanation and expertise.



We covered video delivery to mobile devices in Chapter 10.

MIME Types

MIME types, many of which are listed in Appendix A, are a key element for content delivery. Mobile browsers don't care about the file extension; they decide whether or not to accept the content based on the MIME type delivered by the server. Remember that the MIME type travels with the HTTP header response.

Static Definition

The simplest way to define the right MIME types is to statically define them on your web server. If you are working with a shared hosting service, the control panels often allow you to define document MIME types. If you manage your own server, you can set them up with the following instructions.

Apache

In Apache, the simplest way is to open the *mime.types* file located in the *conf* folder of the Apache root. In your favorite text editor, you can add one row per MIME type to be configured.

You will find hundreds of MIME type declarations. The first thing to do is to look for the following line and change the MIME type to the correct one for mobile XHTML documents:

```
text/html          html htm
```

As you can see, each line contains a MIME type followed by a series of spaces or tabs and a space-separated list of file extensions.



This technique applies these changes to all the websites on the server. If you want to make changes to only one website or one folder of a website, you should create or edit the *.htaccess* file in the appropriate folder and use the AddType procedure:

```
AddType text/x-vcard vcf
```

You can find an Apache configuration file to download and use with all the important mobile web MIME types at www.mobilexweb.com/go/mime.

Internet Information Server

Configuring MIME types in Microsoft IIS can be done via the UI (as opposed to in Apache, where you need to edit a text file).

In IIS from version 7.0:

1. Go to the IIS Manager.
2. Navigate to the level you want to manage.
3. In Features View, double-click on MIME Types.
4. In the Actions pane, click Add.
5. Type the file extension and MIME type and press OK to finish.



In IIS, you can also manage MIME types from the command line. Check the documentation for more information.

In IIS 7.0, it is also possible to define static MIME type declarations in the *web.config* file in your ASP.NET folder. The syntax is:

```
<configuration>
  <system.webServer>
    <staticContent>
      <mimeTypeMap fileExtension=".mp4" mimeType="video/mp4" />
      <mimeTypeMap fileExtension=".vcf" mimeType="text/x-vcard" />
    </staticContent>
  </system.webServer>
</configuration>
```

```
</staticContent>
</system.webServer>
</configuration>
```

Dynamic Definition

The other possible way to declare MIME types is to use dynamic header declarations in your server script code.

In PHP, you should define the MIME type before any other output, using the `header()` function:

```
header('Content-Type: application/xhtml+xml');
```

If you are delivering downloadable content (not markup), like a video, you should also define a filename. If not, when the file is saved it will have a *.php* extension and it will not work.

To define the name of the file we use the `Content-disposition` header, as shown in the following sample:

```
$path = '/videos/video.mp4';
header('Content-Type: video/mp4');
header('Content-disposition: attachment; filename=video.mp4');
// We serve the file from our local filesystem
header("Content-Length: " . filesize($path) );
readfile($path);
```

ASP.NET has a `Response.ContentType` property that we can define:

```
// This is C# code
Response.ContentType = "application/xhtml+xml";
```

The filename should also be defined if it is downloadable content, using `Response.AddHeader`.

In a Java servlet or JSP, you should define the headers using the `setContentType()` method of the response object:

```
response.setContentType("application/xhtml+xml");
```



When you are serving nonmarkup content using a dynamic script, if an error occurs you will not see the error details and the content will be broken (imagine a JPEG with a PHP error as the contents). You should capture any error, send yourself an email or log the error details, and replace the output with generic content.

File Delivery

To deliver a file, there are three models:

- Direct linking
- Delayed linking
- OMA Download

You can use any of these three methods to deliver the files, either using the physical file (video, audio, game, etc.) directly or via a script (PHP, ASPX, etc.). If you use a script to deliver a file, you can log, secure, and even charge for every download. If the file is available directly through the web server, anyone with the URL can download the file.



The installation of files using HTTP is also called OTA (over-the-air) provisioning. Some low-end devices don't have a web browser but do have the ability to download files (e.g., ringtones, applications, or images) using HTTP. We can offer files to those devices, but we must send the download URLs by WAP Push using SMS.

Direct Linking

Direct linking is the simplest way to deliver content. A direct link is just a link to the file (with the right MIME type defined), a link to a script that will deliver the file, or a link to a script that will redirect the user to the file. For example:

```
<a href="game.jad">Download This Game</a>
<a href="download.php?id=22222">Download This Game</a>
```

The *download.php* script can save the download to the database, check permissions, and then deliver the content using the appropriate MIME type, writing the file to the response output or redirecting the browser to the file:

```
<?php
if ($everything_ok) {
    header('Location: game.jad');
} else {
    header('Location: download_error.php');
}
```

Delayed Linking

Delayed linking is a technique often used in download sites for desktop browsers. It allows us to show a landing page before the download starts. This landing page will also be the document the user will see after the download has finished or, if the browser supports background downloading, while it is downloading.



Some devices also look for the `type` attribute in a link to decide how to manage the link before downloading the response from the server. For example, we can define a link as a video file using:

```
<a href="video.mp4"
  type="video/mp4">Download This Video</a>
```

The technique involves linking to an HTML document that will show the user some information and will use a `refresh` meta tag to redirect the user to the direct link in *x* seconds (more than 5 for mobile devices).

So, the download page will redirect to:

```
<a href="download.html">Download This Game</a>
```

And *download.html* will contain code like the following:

```
<!DOCTYPE html>
<head>
<meta charset="UTF-8" />
<meta http-equiv="refresh" content="5;document.pdf" />
<title>Download File</title>
</head>

<body>
<h1>Your document is being downloaded</h1>
<p>If the file is not downloaded in 5 seconds,
  <a href="document.pdf">click here</a>

<h2>Brought to you by "Your favourite ad here"</h2>
<a href="/">More Downloads</a>
</body>
</html>
```

OMA Download

OMA Download is a standard defined in 2004 by the Open Mobile Alliance to allow us more control over the delivery of media objects. It also has support for Digital Rights Management (DRM) in two versions, OMA DRM 1.0 and OMA DRM 2.0. The specification was used as the basis for the Java ME MIDlet OTA installation method that we will see later in this chapter, so the two are very similar.

Unfortunately, modern platforms—including Safari on iOS and Android browsers—don't support OMA Downloads, and we can use them only on platforms that were on the market before 2008, such as feature phones, older BlackBerry devices, or Symbian devices.



Many browsers support the download of Multimedia Messaging Services (MMS) templates using a format called the *Synchronized Multimedia Integration Language* (SMIL). This is useful if you are offering templates like postcards. You can create messages using Nokia tools (<http://www.mobilexweb.com/go/mms>).

OMA Download adds two phases to the download process: *before download* and *after download*.

The before-download process involves a description file downloaded using HTTP before the real file is downloaded. This description file is an XML file containing meta-information for the operating system and instructions for doing the download. This process gives the user an opportunity to see information about the content (name, compatibility, size) before accepting it.



Using OMA DRM you can protect a file from being sent by MMS, Bluetooth, or any other method after it has been installed on the device. This is to avoid piracy of video, music, Flash Lite, and other multimedia content. Nokia, Sony Ericsson, and Motorola devices are known to have support for this standard.

The after-download process involves an HTTP request posted to your web server from the operating system, confirming the final status of the download. This allows you to confirm that the file has been correctly downloaded and installed.

OMA DRM supports the ability to define the right to play, display, or execute a media object or a file a limited number of times. DRM should be managed with care, and you need to test compatibility with the devices you're targeting before using it.

Download descriptor

The descriptor is an XML-based file that should be served using the MIME type `application/vnd.oma.dd+xml`.

Here's a simple example of this file:

```
<?xml version="1.0"?>
<media xmlns="http://www.openmobilealliance.org/xmlns/dd">
  <name>The first man on the moon</type>
  <type>video/mp4</type>
  <objectURI>http://mobilexweb.com/video.mp4</objectURI>
  <size>230</size>
  <installNotifyURI>http://mobilexweb.com/download/notify.php?id=3333
  </installNotifyURI>
</media>
```

These are the typical attributes: `name`, the user-readable name of the content file; `type`, the MIME type of the file; `objectURI`, the absolute URL of the file; `size`, the file size expressed in KB; and `installNotifyURI`, the URL that will receive the after-download status (this can have GET parameters defined dynamically to log whether or not the download was saved).

Additional properties are also available, like `nextURL` (the URL of a website to visit after the download has finished), `description` (a short description of the media file), `vendor` (the organization providing the file), and `iconURI` (an optional icon to be shown with the file information).



There are good resources and tools on the Nokia Developer, Sony Ericsson, and Adobe websites about OMA Digital Rights Management (<http://www.mobilexweb.com/go/drm>).

Post-download status report

If you define the `installNotifyURI` in the download descriptor, you will receive a POST request to that URL when the download finishes. This URL will receive as the POST body an integer code with a status message. The important thing is that this request does not come in the normal URL-encoded way, so you can't use the typical `$_FORM` or `Request.Form`. To read the status code, you'll need to read the request body in a low-level format. Table 17-1 lists the most common OMA Download status codes to read in our scripts.

Table 17-1. OMA Download status codes

Code	Message	Description
900	Success	The object was downloaded and installed.
901	Insufficient Memory	The device has no space to download or install the file.
902	User Cancelled	The user cancelled the download.
903	Loss of Service	The device lost the network connection while downloading the file.
905	Attribute Mismatch	The file doesn't match the download descriptor (e.g., the MIME type). The file will be rejected.
906	Invalid Descriptor	The download descriptor is invalid.
951	Invalid DDVersion	The download descriptor version is invalid.
952	Device Aborted	The device aborted the installation process. This can occur for different reasons.
953	Non-Acceptable Content	The device cannot use the file.
954	Loader Error	The URL of the file is not working.

To read the OMA Download response from a PHP script, you can use the following sample code:

```

<?php
// We get the post body from the input
$post = file_get_contents('php://input');
// We get the first three characters without spaces
$status = substr(trim($post), 0, 3);

if ($status==900) {
    // Download OK, save information to the database
} else {
    // Some error happens, save information to the database
    // to allow the same user to download it again
}

?>

```

If you are delivering premium content that the user has paid for, if the download fails you should deliver the same content without forcing the user to pay again. Even if the download has succeeded, many carriers insist that the content be made available free of charge for 24 hours (or even up to a week). Some new application stores also allow the users to download the premium content again even if they change their mobile devices.



Remember that you can check for OMA Download compatibility using WURFL with the `oma_support` capability before using this download mechanism.

Application and Games Delivery

In the beginning, the mobile content delivery world centered around ringtones and wallpapers. To deliver this kind of content, we should rely on what we have seen before: if we deliver the proper MIME type, the file will be saved by the mobile device.

Application delivery can be useful:

- If you are creating a game or application store
- If you are developing a mobile website for a current application
- If you have a richer version of your mobile website available as a widget or application
- If you are providing a shortcut for your website embedded as an application

The application formats that you can deliver from a website are:

- Java ME (formerly J2ME)
- Flash Lite
- Android (if the user allows installation from nonsecure sites)

- Symbian native
- Widgets
- BlackBerry (up to 7.1)

We cannot deliver native iOS, BB10, Windows Phone, or Windows 8 Store applications for end users, as the specific platform store is the unique public way to install and deliver applications for each of these OSs. However, we can provide a link on the screen to the native store application that we want the user to buy or download.



Always verify, on the server or the client side, that the user is on a compatible device before delivering applications or links to application downloads.

iOS Applications

If you have your own application that has already been accepted for distribution via the App Store, or if you want to provide users with a link to buy or download an application, game, ebook, music file, movie, or TV show, you can use a special iTunes link that will open iTunes or the App Store automatically, displaying the desired content.

You can create one of these links using the web service iTunes Link Maker (<http://www.apple.com/itunes/linkmaker>). You can select which country's App Store to look for the content in, and then search for the content you want to link to.



From iOS 6 you can also deliver PassBook (<https://developer.apple.com/passbook>) files, including passes, coupons, and virtual cards.

For example, to provide a link that the user can visit to buy the movie *Terminator Salvation*, we can use the following code provided by the iTunes Link Maker:

```
<a href="http://itunes.apple.com/WebObjects/MZStore.woa/wa/viewMovie?
id=338372479&s=143441&uo=6" target="itunes_store">

</a>
```



In Chapter 7, we talked about iOS Smart App Banners, a way to connect a website with a related native app in the App Store that will also invite the user to open the app if it's already installed.

Ad hoc OTA installation

Earlier I said that iOS applications can't be installed from a website, and they have to be downloaded from the App Store. That was not completely true, as there are some exceptions that will allow us to install an app from a website; this is called *ad hoc distribution* or OTA (over-the-air) installation.

We can distribute an app from a website in the following situations:

- We have a Corporate iOS Developer Account that allows us to create internal apps for our company.
- We have a Standard iOS Developer Account (personal or business) and we want to deliver the app to beta testers (up to 100).
- We have a Standard iOS Developer Account (personal or business) and we want to deliver the app to our own devices (up to 100).

In all cases, the app needs to be signed with a specific signature and mobile provisioning file that defines one by one the devices where the app can be installed.

After the compilation process you will have an *.ipa* file (the iPhone Package application) and a *.mobileprovision* file (the counterpart that needs to be installed on the device, which allows installation of nonstore apps).

The mobile provisioning certificate should be delivered from a website as a normal file (no specific MIME type is available, *application/octet-stream* works), but the *.ipa* file needs special attention. If you link to the *.ipa* file directly, it won't work. Instead, you need to create an intermediate download file called the *download manifest*—a similar concept to OMA Download.

The download manifest is an XML file in *plist* (property list) syntax. You can create this file by hand or using Xcode when you do the final archive compilation, with the option "Save for Enterprise Distribution." The manifest file will include the final IPA absolute URL inside.

To link to the download manifest file, you need to use the *itms-services://* protocol and the absolute URL to that file, as in:

```
itms-services://?action=download-manifest&url=<absolute_manifest_url>
```

When the user clicks on a link with this URL, Safari will ask for permission to install the app. If the user accepts, and if he already has the mobile provisioning certificate installed, the app was compiled with it, and the current device is on the list of approved devices for OTA installation, the app will finally be installed on the iOS device, bypassing the App Store.



There are some cloud-based services that will help you install ad hoc apps, such as TestFlight (<http://www.testflightapp.com>), without configuring your own server. Alternatively, HockeyKit (<http://hockeykit.net>) is an open source framework to provide OTA installation for iOS and Android users from your own server.

Android Applications

Android devices may have different stores for apps, such as the Google Play Store (formerly known as Android Market) and the Amazon Appstore for the Kindle Fire and other devices.

If you make an HTTP-based URI to play.google.com, the user will be asked if she wants to open the website or the store app, showing on that content. The best way to link to the details of a native published app is to use the https://play.google.com/store/apps/details?id=<package_name> URL. The `<package_name>` should be replaced with the package name that you defined for your app when creating it. We can also use the [market://details?id=<package_name>](https://play.google.com/store/apps/details?id=<package_name>) URI mechanism.



On Android 2.x, it was common to use the `market://` protocol to connect to the Android Market, now the Google Play Store. Today, we can use both the <http://play.google.com> and `market://` URIs. The second one will always open the store, while the first one should ask the user what to do.

We can also list all the apps that a publisher has online using http://play.google.com/store/search?q=pub:<publisher_name>. For more options, visit the Android developer site (<http://bit.ly/ZQi7Zj>).

If we have an app published in the Amazon Appstore, we can use the following URI template: http://www.amazon.com/gp/mas/dl/android?p=<package_name>. The Appstore app will be opened if it's installed on the device.

Bypassing the store

Android devices allow app installation from third-party servers if the user has the feature enabled from settings (the menu option varies per version). To deliver an Android application, we just need to serve the `.apk` (Android package) file as `application/vnd.android.package-archive`.

Windows Applications

For Windows Phone applications, we can make a link to *http://windowsphone.com/s?appId=<GUID>* that will redirect the user to the specific app in the store. The *<GUID>* is a placeholder for the ID of the application we want to link to.

The ID of your application is a long hexadecimal value; the easiest way to get the full link is to access your account in the Windows Phone Dev Center, go to your app's details, and find the link under the section "Deep Link."



For other platforms, usually making an *http://* link to the app's details page will trigger the native store application to download that app.

For Windows 8 applications, we can use the *ms-windows-store:* protocol, following the syntax *ms-windows-store:PDP?PFN=<package_name>*. You can find the package name in your project's properties in Visual Studio or by visiting your app's web-based details page and looking at the source code. For example, the ESPN FC app for Windows 8 has a details website at the Windows Store (*http://bit.ly/ZKYU9a*). Digging into the source code, you will find a script with a *packageFamilyName* variable:

```
<script type="text/javascript">

    var autoLaunchStore = false;
    var biEnabled = true;
    var biServer = 'c.microsoft.com';
    var packageFamilyName = 'ESPNCricinfo.ESPNFC_y1atfjxm9t5ma';
    var more='Read more';
    var less='Read less';
    var isDarkText = false;
    var isRtl = false;
    var imageUrl='webpdp/images/'
    var screenshot='Screen shot {0}';

</script>
```

Therefore, to link to this app, we just use:

```
<a href="windows-store:PDP?PFN=ESPNCricinfo.ESPNFC_y1atfjxm9t5ma">
  Download
</a>
```

While digging into the source code feels like hacking the system, it's one of the suggested mechanisms in the official Microsoft documentation.



For Windows 8 apps, remember that we can add some meta tags so IE10 will invite the user to install or open the native app that is connected to the current website. We covered this ability in Chapter 7.

Java ME

Java ME was the preferred language for games and applications for years. Its usage is declining today, but it is still the platform behind millions of Series 40 devices, Symbian devices, and BlackBerry smartphones (before BB10). Even Facebook and Angry Birds have versions of their apps in Java ME today, for emerging markets.

A Java ME project is shipped as a *JAR* (Java ARchive) file, which is just a ZIP file containing the application (compiled classes and resources). It must be delivered using the MIME type `application/java-archive`. Many phones accept this file type directly, although the best (and 100% compatible) way of delivering Java ME games or apps is to first deliver a *JAD* (Java Application Descriptor) file. The JAD file is just a text file served with the MIME type `text/vnd.sun.j2me.app-descriptor` that contains metadata about the application, similar to OMA Download's download descriptor files (in fact, OMA Download took this approach from Java).

So, the device first downloads the JAD file and shows the information to the user (name of the application, size, format, etc.). If the user accepts, the JAR file is then downloaded and installed. The Java ME developer usually generates the JAD file, and we receive it in its final state. However, as it is a text file, we can generate it ourselves or change it using a server-side script.

Serving JAD files

Let's analyze a part of the MIDP 1.0 version of the JAD file sent to the device when a user tries to download Opera Mini 3:

```
MIDlet-Version: 3.1
MIDlet-1: Opera Mini 3, /i.png, Browser
MIDlet-Data-Size: 10240
MIDlet-Description: Opera Mini
MIDlet-Icon: /i.png
MIDlet-Info-URL: http://mini.opera.com/
MIDlet-Install-Notify: http://mini.opera.com/n/13045Bviprdome_en
MIDlet-Jar-Size: 58800
MIDlet-Jar-URL: opera-mini-3.1.13045-basic-en.jar
MIDlet-Name: Opera Mini 3
MIDlet-Vendor: Opera Software ASA
Content-Folder: Applications
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-1.0
```

The emphasized parts of the code are the ones that we need to care about when delivering Java ME applications.



There are a lot of other standard and vendor-specific JAD attributes that can be defined, from virtual keyboard support on touch devices to digital signatures.

`MIDlet-Jar-URL` defines the relative or absolute URL of the JAR file. We can insert the JAR file directly here, or, if we want to secure and log the download, we can use a URL to a dynamic script using a URL parameter included in the JAD generation.

`MIDlet-Install-Notify` is the same as the `installNotifyURI` parameter in OMA Download. It is an optional parameter that defines a URL that will receive by POST the same codes as in OMA Download (from 900 to 906), as seen in Table 17-1.



There is another optional JAD attribute, `MIDlet-Delete-Notify`, that defines a URL that will receive by POST a notification when the user deletes the application from the device. Using this attribute is not recommended; it is not reliable, and the user may not want to connect to the Web when deleting an application.

Starting with MIDP 2.0 (the version compatible with almost all Java ME devices on the market today), the standard added new codes that we can receive in the `MIDlet-Install-Notify` URL. The added status codes are shown in Table 17-2.

Table 17-2. Additional MIDP 2 status codes

Code	Message	Description
907	Invalid JAR	The JAR (executable package) is invalid and could not be installed.
908	Invalid Configuration or Profile	The device is not compatible with the versions of the libraries used in the package.
909	Application Authentication Failure	A security problem has occurred.
910	Application Authorization Failure	A security problem has occurred.
911	Push Registration Failure	A push notification registered in the JAD file is invalid.
912	Deletion Notification	The user has deleted the application from the device (used when <code>MIDlet-Delete-Notify</code> was defined).
913	Required Package Not Supported by the Device	A package or API marked as required by the application is not available on the device.

Java ME for BlackBerry

BlackBerry smartphones (not including those using the new BB10 platform) accept the same JAD and JAR files that we've been examining. However, the most compatible way to serve Java ME files on these devices is to use BlackBerry's own format for JAR files: *COD* files.

BlackBerry uses the same JAD files, with two new mandatory attributes: *RIM-COD-URL* and *RIM-COD-Size*. The COD file must be served as *application/vnd.rim.cod*, and it is generated using a free tool from BlackBerry that converts a JAR into a COD file (<https://developer.blackberry.com/java/>).

Debugging and Performance

Debugging and performance optimization are the two scariest activities in the mobile web development world, after testing—but don't worry, there are lots of ways to tame them. We discussed tools and techniques for testing in Chapter 4. In this chapter we will focus on more advanced features for debugging and performance.

Debugging

For years I've complained that web developers were second-class citizens (compared with native developers), in terms of the availability of tools and documentation. Fortunately, in the last year or so vendors have begun to realize that web developers need more tools for debugging and performance measurement, and the tools catalog available today is growing.

Server-Side Debugging

To debug server-side detection, adaptation, or content delivery scripts, we can use some HTTP tools before turning to the real devices.

User agent spoofing

User agent spoofing tools allow us to fool the server about the browser that is currently requesting the web page by changing how the client identifies itself to the server. For example, using a desktop browser, we can say it is a mobile browser and the server will fall into the trap. We can use these tools to debug mobile redirects and detection frameworks, as covered in Chapter 9, without the need of real devices or even emulators.

We can then browse to any website and see how the server manages the user agent and which content it serves. (Remember to go back to the default user agent after finishing the debug session, or you may encounter problems in your browser!)

User Agent Switcher (<http://www.mobilexweb.com/go/uaswitcher>) is a free plug-in for Firefox (see Figure 18-1).



Figure 18-1. With User Agent Switcher you can test websites using any mobile user agent and render them with Firefox for desktop.



User agent tools don't emulate all the headers of a mobile device, including the accepted MIME types, so you should not rely on this plug-in for testing this kind of detection. You can use other plug-ins to change HTTP headers.

When you've installed this plug-in, you will find a new submenu in the Tools menu of Firefox using the name of the current user agent (it starts with "Default User Agent"). The plug-in changes the user-agent string that Firefox uses for making HTTP requests to the server. It comes with some user agents preinstalled, such as iPhone 3.0, and you can add as many others as you want using the Edit User Agent option.



Keynote MITE (<http://mite.keynote.com>) (Mobile Interactive Testing Environment) is a simulator that can be useful for mobile browser user agent spoofing and HTTP header debugging.

Google Chrome for desktop includes the Chrome Developer Tools, a series of debugging tools for web developers. The latest versions of the desktop browser also include some tools for mobile web debugging, including a user agent cheating mechanism. To use them, go to View→Developer→Developer Tools, access the Settings (the small gear icon at the bottom-right corner), and browse to the Overrides section, as seen in Figure 18-2.



Figure 18-2. Google Chrome for desktop includes tools for changing the current user agent and matching device metrics with the page's available viewport.



Opera and Safari for desktop also offer native features to change the user agent without any plug-ins required, using developer tools.

HTTP sniffing

When using real devices, it will be useful while debugging to store in some log all the request and response headers from the server-side code, so you can see the data the device is sending and receiving. DeviceAnywhere (<http://www.keynotedevicewhere.com>) includes a solution for this purpose for all devices, or you can use any emulator that supports HTTP sniffing, like the Nokia and BlackBerry emulators.

With other emulators and simulators, as well as with some real devices, we can use the typical desktop HTTP sniffing tools, such as Telerik Fiddler (<http://www.fiddler2.com>) or Charles Web Debugging Proxy (<http://www.charlesproxy.com>).



Some mobile web browsers act differently when connected directly to the Internet than when connected through an HTTP proxy. Therefore, be careful when making conclusions based only on a mobile browser connected through a desktop HTTP proxy.

If you want Android browsers to browse the Web through a proxy (usually connected to a WiFi router and a proxy in the same LAN), you can define this behavior through the desired WiFi connection's "Modify network" contextual menu, activating Advanced Options and selecting Manual Proxy, as seen in Figure 18-3. On iOS devices you need go to Settings→Wi-Fi, edit the hotspot using the blue edit action button, and select Manual HTTP Proxy. On other operating systems, the process is similar.



If you work with the ASP.NET platform on the server, you can activate the remote tracing mechanism and you will see every header and response from your mobile devices.

Bandwidth simulators

With bandwidth 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.

The Charles Web Debugging Proxy (<http://www.charlesproxy.com>) 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 18-4). It is available for Windows, Mac, and Linux.

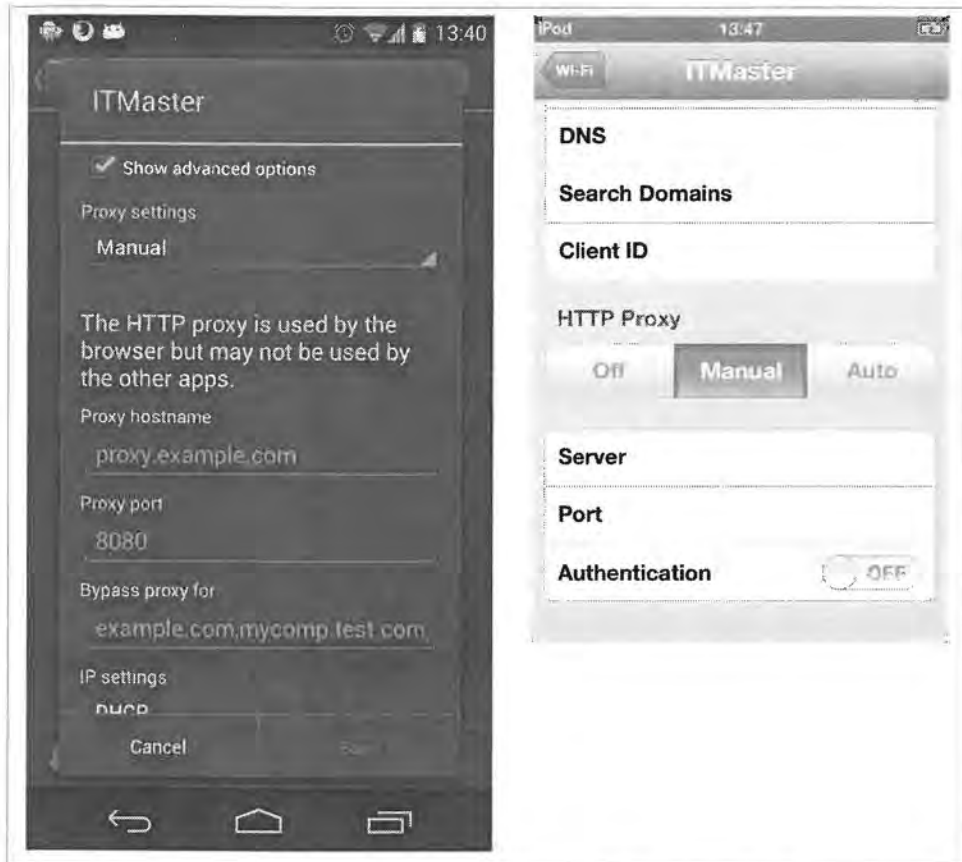


Figure 18-3. On Android and iOS devices we can define an HTTP proxy for web browsers through the WiFi hotspot advanced settings.

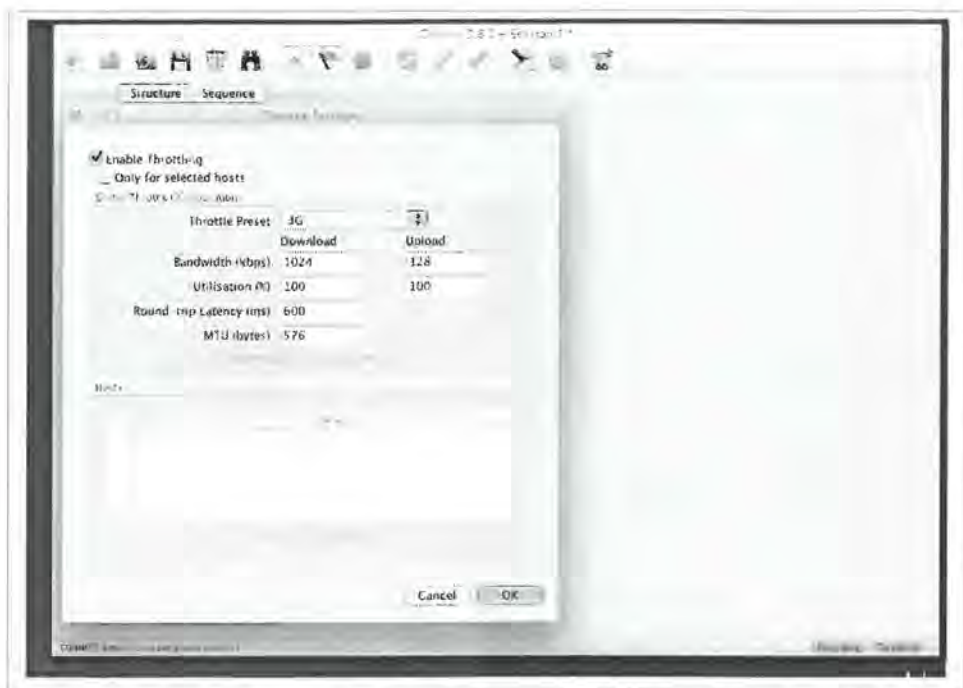


Figure 18-4. The Charles Web Debugging Proxy allows us to simulate a real bandwidth through the Proxy Throttling feature.



On Mac OS, Xcode (from 4.1) includes a System Preferences pane called *Network Link Conditioner* that simulates predefined bandwidth, latency, and packet loss on the whole operating system.

Slow App (<http://slowyapp.com>) is a Mac-only app that will simulate a real-world connection. It includes several presets, including bandwidth, packet loss rate, and packet latency. Slow is not a proxy, so when using Mac-based simulators and emulators we don't need to configure any proxy. NetLimiter (<http://www.netlimiter.com>) is a Windows-only app for the same purpose. It includes a free limited version.

Fiddler Connection Simulator (<http://mobilexweb.com/go/fiddlersimulator>) is a Fiddler proxy plug-in adding the ability to simulate different network connections on Windows computers.

Markup Debugging

There is no automatic way to debug HTML. This is a manual operation on every emulator, device, or remote device you can access. Some browsers include a remote de-

bugger that we can check for markup errors (we'll cover them in a minute), but before doing this it is a good practice to validate the code using one of the online tools available for mobile markup.

HTML5 validators

If you want to validate your HTML5 code, you can find some beta tools such as W3C's Nu Markup Validation Service (<http://validator.w3.org/nu>) and the Markup Validation Service (<http://validator.w3.org/nu>). However, remember that HTML5 is not finished yet, and there are some extensions outside of the specific standards—therefore, these tools will be useful primarily for getting a big picture of errors and warnings. You should check these one by one to see if they're actually errors or nonstandard features you want to use.

W3C mobileOK Checker

The W3C offers a mobile markup checker (<http://validator.w3.org/mobile>) that you can use for free. You can upload a file, copy and paste the code, or use a URL if you already have your mobile site on your server.

This markup checker is based on best practices published in the Mobile Web Best Practices standard (<http://www.w3.org/TR/mobile-bp>). It doesn't guarantee that your code will work perfectly on all mobile devices if it passes; it is just intended to help you find possible problems in your code and areas that don't conform to best practices. This tool is intended for XHTML, XHTML MP, and XHTML Basic websites, rather than modern HTML5 websites; however, it will still give us some hints about good practices to follow and potential errors we should investigate.

ready.mobi

The dotMobi team has created a free validator that includes the W3C mobileOK Checker tests and some others, plus some emulators and detailed error reports with suggestions. The validator is available at mobiReady (<http://www.ready.mobi>). You can use it for a single document by providing a URL or copying and pasting the code, or to report on an entire site, including site-wide testing (registration is required for this last function).

After analyzing your document, ready.mobi will assign you a score on a scale of 1 (very bad) to 5 (excellent). It will also report on the size of your document and resources and the estimated time and download costs for the user.

This report is only suitable today for feature or social phone websites, not for HTML5 web apps.

Client-Side Debugging

Client-side debugging is one of the most painful activities in mobile web development. Every browser has a different rendering and JavaScript engine, and sometimes code that works on one device doesn't work on another.

Typical desktop JavaScript techniques should be used first to debug logic problems in our code. This includes using the developer tools included with Chrome, Safari, or Internet Explorer, or the classic Firebug (<http://getfirebug.com>) for Firefox. But just because everything works in a desktop browser doesn't mean that it will work in a mobile browser.

The debugging solutions in this area include different kinds of tools:

Remote debuggers

These open a debug session on the device that we can access remotely from a desktop computer, through WiFi or a USB cable. These tools can also be divided into official browser debuggers and injected debuggers, with the latter being a JavaScript file we need to inject in our HTML to provide the debugging session.

On-device debuggers

Here the debug session is seen on the device itself, in a separate window or pop-up.

Console output viewers

These are used only to see console output messages and errors from the browser.



One problem we will have is that if a JavaScript error is encountered, many devices don't show any notice and the code simply ends its execution.

iOS Remote Web Inspector

To debug web apps for iOS devices we can use the Remote Web Inspector, available since iOS version 6. The remote session can be connected from Safari 6 for Mac to the iOS simulator or a USB-connected iOS device.

We can open a web remote inspection session to:

- A Safari window/tab on our iOS device or simulator
- A chromeless web app installed on our iOS device or simulator
- A native app using a web view, such as an Apache Cordova/PhoneGap app (this feature is available only for our own native apps; we can't debug an app that was installed from the App Store)



Unfortunately, at the time of this writing there is no way to open the iOS web debugger from a Windows or Linux desktop computer. We must rely on other nonnative solutions if we need to debug from a non-Mac desktop computer.

Connecting the session. To use the Web Inspector on iOS 6.0 or newer, we must first enable it from the iOS device or simulator itself, by going to Settings→Safari→Advanced and turning on the Web Inspector feature, as seen on the left in Figure 18-5. The next step is to enable the developer tools in Safari for Mac, via Preferences→Advanced→“Show Develop menu in menu bar.” When we’ve done all this and the session we want to debug is open on the device or simulator, we can access the Develop menu in Safari for Mac; there we’ll see the name of our device as a submenu with all the available web sessions to connect to on it, as seen on the right in Figure 18-5.



Figure 18-5. To use the Remote Web Inspector on iOS, we need to first enable it from the device or simulator (left), and then use it from Safari for Mac (right).

Working with the session. When we’re connected, Safari for Mac will open a window with our debug session user interface (as we can see in Figure 18-6). In the debug session we will be able to:

- Inspect the HTML, CSS, and JavaScript
- Make changes and see them live, without a refresh
- Touch to inspect (the little hand icon at the top bar)—i.e., we can touch an element on the device to select it on the remote DOM
- Browse storages, such as cookies, local storage, SQL databases, and more

- Record and measure timelines, including network requests, layout and rendering, and JavaScript execution and events
- See warnings and errors
- See web workers (threads)
- See and manage JavaScript breakpoints, including watch expressions
- See the console and execute JavaScript directly on it



Figure 18-6. The Remote Web Inspector includes a debug UI similar to Xcode, the IDE for native iOS and Mac applications.

Older iOS debugging tools

Previous to iOS 6 (up to mid-2012), the only official debugging solution was a Console view that could be enabled from Safari settings, allowing the developer to see the console and error messages on iOS 3, 4, and 5 devices. This feature was on-device and it was deprecated in iOS 6, as the full Web Inspector replaced it.

In iOS 5, a private API was discovered that opened a Web Inspector session in the iOS simulator only. Using this trick, I created a free tool called iWebInspector (<http://iwebinspector.com>) that allows developers to open a Web Inspector session and use it easily. This tool is still available if you need to debug iOS 5 web applications, but you should use the official remote inspector for newer devices.

Chrome for Android debugger

Google Chrome for Android supports remote debugging through a USB connection and the Developer Tools available in Chrome for desktop. To use the debugger, you need to have the Android SDK installed on your computer (see Chapter 4 for details). Then, follow these steps:

1. Enable USB debugging on your Android device (under Settings→Developer options).
2. Connect your mobile device to your desktop using USB. If your host is a Windows-based computer, you will need debug USB drivers from the manufacturer.
3. Open Chrome on your Android device and go to Settings→Advanced→Developer tools. Check the “Enable USB Web debugging” option.
4. Open a terminal or command-line session on your host and execute the following command line from the SDK path's *platform-tools/* subfolder:

```
adb forward tcp:9222 localabstract:chrome_devtools_remote
```
5. Open Chrome on your desktop computer and browse to <http://localhost:9222>.
6. Select the mobile tab you want to debug.

When connected you can use the Chrome Developer Tools as usual, to do all of the following:

- Inspect elements
- Access the resources list, including local storage, SQL, and IDB databases
- Access network information
- Add JavaScript breakpoints and debugging tools
- See useful timelines, such as rendering information
- Profile JavaScript code
- See the console and execute JavaScript code on it

Check out more information and tips about the Chrome Developer Tools (<https://developers.google.com/chrome-developer-tools>).

Firefox remote debugging

In Firefox 15 and later, the desktop browser has the ability to connect to a mobile version of the same browser. To use this feature, you need to have Firefox installed on both your desktop and a mobile device. Then, follow these instructions:

1. Open *about:config* in Firefox for desktop and define `devtools.debugger.remote-enabled` as `true`.

2. Open *about:config* in Firefox for mobile and define `devtools.debugger.force-local` as `false` and `devtools.debugger.remote-enabled` as `true`.
3. Restart Firefox on the desktop and the mobile device.
4. Connect your mobile device (using WiFi) to the same network as your desktop computer.
5. Find your current mobile device's IP address (go to Settings→WiFi and tap on your current WiFi session).
6. Browse to the desired page on your mobile.
7. In Firefox for desktop, open the Remote Debugger from the Web Developer menu.
8. Replace *localhost* in the new window with your mobile IP address.

BlackBerry remote Web Inspector

The BlackBerry browser for smartphones (7.x), PlayBook, and the newest BB10 platform supports a remote Web Inspector through WiFi. You can enable it through the browser's settings, where you will receive the URL to use on your desktop to open the Web Inspector session.



When enabling remote sessions using WiFi, as with the BlackBerry browser or Firefox, there is a security risk, as everyone on your own LAN can access the content, cookies, and storages on your mobile browser.

Opera Dragonfly

From Opera Mobile 9.5, we can debug mobile web applications using the remote debugging tool Dragonfly. To use this tool you will need Opera 9.5 or later (with Presto engine) on your desktop. You can open Dragonfly by going to Tools→Advanced→Developer Tools and checking the Remote Debug option.

When you're done, enter *opera:debug* in the address bar of your Opera Mobile browser and specify your desktop IP address (public or private, if you are connected using WiFi to the same LAN). You will then have access to the same debugging features (DOM, CSS, and JavaScript) that you would if you were debugging a local desktop file.

Complete instructions and tips can be found at www.mobilexweb.com/go/dragonfly.

Android Debug Bridge

While the Android browser doesn't have a remote debugger (at least, as of Android 4.2), we can still read the console errors and even use the same console object using the

Android Debug Bridge (*adb*). *adb* is a command-line application available in the *tools* folder of your SDK.

You can find more information on how to use this console at mobilexweb.com/go/adb.

Weinre

Weinre (<http://people.apache.org/~pmuellr/weinre/>) (read it as “winery”) is an open source tool now part of Apache Cordova that implements a Remote Web Inspector based on WebKit developer tools. Weinre was created before any platform officially supported a remote inspector, and it is implemented as a server (now a Node.js app; previously a Java servlet) and a debug client (JavaScript code that we need to inject into our HTML).

Because the whole inspection is done directly with the JavaScript code we inject, the inspection session is limited compared to native ones: we can't use breakpoints and there are no profiling abilities.



As there is no official remote debugger or inspector for the Android browser at the time of this writing, Weinre is the best solution available.

To install it, you can follow the instructions on the website, or you can install Adobe Edge Inspect (covered in Chapter 4), which includes a Weinre server. PhoneGap also provides a free-to-use Weinre server (<http://debug.phonegap.com>). To use it, you need to define a unique ID of your own and inject the following script into your HTML page:

```
<script  
src="http://debug.phonegap.com/target/target-script-min.js#<id>">  
</script>
```

When you're done, accessing <http://debug.phonegap.com/client/#<id>> from a desktop will open the remote session. (Remember to replace *<id>* with your own unique ID.) The advantages of using this version are that we can debug devices without any other inspection tool, and we can use it through 3G. The main disadvantage is performance, as all the debugging information needs to go through the PhoneGap server on the Web. Installing your own server will improve the performance of the debugging session.

JSConsole

JSConsole is a free online command-line tool created by Remy Sharp (<http://remysharp.com>) that can be used directly on the device, by accessing jsconsole.com from the mobile browser, or remotely, for usability purposes. To create a remote session, you need to first open JSConsole (<http://jsconsole.com>) in your desktop browser and then type `:listen` in the console. You will get a long unique ID that you can use to inject the following script into the HTML page you will load in the mobile browser:

```
<script src="http://jsconsole.com/remote.js?<ID>"></script>
```

Remember to replace `<ID>` with the ID received in your desktop console. When you are connected, you can execute commands from your desktop on the mobile device.



When injecting debugging session scripts, such as in Weinre or JSConsole, remember to remove those references when you go to the production environment.

Bookmarklet solutions

Bookmarklets give us access to tools and techniques for testing and debugging that are not typically available on mobile devices. We can use them to access resource lists, inspect source code, and much more. We covered the basics of bookmarklets in Chapter 12. One of the most useful bookmarklet toolkits is Mobile Perf (<http://bit.ly/YI40UG>) by Steve Souders (see Figure 18-7); it gives us easy access to a wide range of tools for debugging and measuring and working with performance.

Other web-based solutions

There are some scripts that work as a kind of debugger, including DOM and CSS inspectors, and some that work for JavaScript debugging, too. The mobile compatibility for these tools is complicated, though, because of the lack of space on the screen to show all the information. There are also some Ajax-based solutions that will work better, allowing us to view the debug results and panes from a desktop.



Using alert windows for logging and debugging is annoying and a bit intrusive. Try to use another solution.

The JavaScript Debug Toolkit (JSDT) (<http://code.google.com/p/jsdt>) is an Ajax-based JavaScript debugging tool that works with mobile devices as a standalone desktop application or an Eclipse plug-in.



Figure 18-7. A bookmarklet (*Mobile Perf* in this example) injects JavaScript code over any page in a mobile browser, giving us access to functionality including debugging tools.



You can check whether `window.onerror` is available and catch every error before blocking all the rest of the script.

Another option is Firebug Lite (<http://getfirebug.com/lite.html>), a plug-in that makes some Firebug tools available in non-Firefox browsers if you add a JavaScript file and a CSS file to your website. It works in many mobile browsers and web app engines, but the navigation is very complicated when the Firebug Lite view is open.

Performance Optimization

Most mobile users expect to make sacrifices in terms of content depth and feature-richness in exchange for the convenience of anytime, anyplace mobile Web access. One thing mobile users won't sacrifice, however, is speed.

—“What Users Want from Mobile”—Equation
Research survey, 2011

Performance is the key to mobile web success. People want high-performance websites. We hate to wait on our desktops, and the situation is far worse on mobile devices, with their constrained resources. I could write a whole book about mobile web performance, but for now I will just try to distill some best practices and share some hacks that you can easily apply to enhance your website's performance.

Performance has recently become a hot topic in the desktop web world. In general, mobile web developers should follow the same practices, but there are some new ones to keep in mind as well, and some desktop web best practices that will not work on these devices.



Web performance optimization is a big topic, and we don't have enough space in this book to cover it completely. If you're just getting started with performance optimization, the first thing you should do is read two excellent books by Steve Souders (<http://stevesouders.com>), *High Performance Websites* and *Even Faster Websites*, both from O'Reilly. You should also follow the Yahoo! and Google performance team blogs at developer.yahoo.com/performance and developers.google.com/speed.

Mobile browsers aren't the same as desktop web browsers, and not all mobile browsers are created equal. Specifically, the quantity of resources that can be downloaded in parallel and the cache functionality differ. Nevertheless, it is better to approach mobile performance optimization from here than from the ground.

If you want to know more about high-performance mobile websites and mobile browser behavior, check out mobilexweb.com/go/performance.

Measurement

The first thing we need to do is to measure. If we cannot measure, we cannot optimize. However, measuring mobile websites is not easy. Typical desktop measurement and profiling tools don't work for mobile devices, and HTTP sniffers are difficult to implement for mobile browsers on 3G networks.

Steve Souders has created a website listing tool (<http://stevesouders.com/mobileperf/>) that will help us in the process.

Remote profilers

All the official remote inspectors or debuggers already covered in this chapter, such as the iOS Remote Web Inspector and the Chrome Developer Tools, include precise tools to measure network and rendering performance and profile our JavaScript code in compatible versions of the browsers.

HTTP proxies

If you are using an emulator or a real device with WiFi capabilities, you can use any HTTP sniffer proxy, configuring the emulator and your device with your desktop IP address and port as the proxy for navigation. There are dozens of tools for doing this, a few of which were mentioned earlier in this chapter. However, bear in mind that browsers may act differently when browsing through a web proxy.

If you have a dedicated server (or even your own development computer with full inbound access to port 80), you can install one of these proxies and a web server and browse your website from any phone on any network to analyze how is it rendering and requesting resources.

PCapPerf (<http://pcapperf.appspot.com>) is a set of open source tools that allow us to capture and analyze traffic generated from a mobile browser through an ad hoc WiFi hotspot created by our desktop computer. When the traffic is captured on the desktop, we can convert the output to the industry HAR standard and display it as a waterfall chart. You can find instructions at the Capture Mobile Traffics wiki (<http://bit.ly/XUdU7K>).

Akamai Mobitest

Akamai Mobitest (<http://mobitest.akamai.com>) is a quick and easy way to get waterfall performance charts from mobile browsers, including on iOS and Android devices. On some devices the test is done through a web view and not the browser, though, so the real performance may differ.

You can start a test by just adding a URL, selecting the device, and choosing the optional feature of recording a video, where you can see how your site is being rendered on a timeline (as seen in Figure 18-8).



Figure 18-8. Akamai Mobitest is a free online service where we can get a waterfall chart of our website being rendered on a mobile browser.

Performance APIs

The W3C's Web Performance Group (<http://www.w3.org/2010/webperf/>) is working on different APIs for accurate performance measurement on the browser. The Navigation Timing API (<http://www.w3.org/TR/navigation-timing/>) is already available in some mobile browsers at the time of this writing, including the Android browser from 4.0, Google Chrome, the BlackBerry browser for BB10, Firefox, and Internet Explorer.

The Navigation Timing API allows us to measure times in an accurate way without affecting the measurement. It exposes a series of timestamps for performance hits, such as navigation start, request fetch start, request end, DOM loading, etc.

Therefore, to calculate how much time has passed from the navigation start point, we can use:

```
var now = new Date().getTime();
var page_load_time = now - performance.timing.navigationStart;
alert("User-perceived page loading time: " + page_load_time);
```

Best Practices

Here are some global best practices you should always have in mind:

- Keep it simple.
- Reduce the HTTP requests to the minimum possible.
- Implement Ajax requests if you can, and if the device supports them.
- Make the cache your friend.



HTTP request headers are generally larger in mobile websites because of the large User-Agent, Accept, and other headers. Remember that these headers are sent with each and every request your page makes. That is why it is important to keep the number of requests to the minimum.

Reducing requests

There are plenty of tips for reducing network requests:

- Use only one CSS and JavaScript external link per page.
- If the script and/or CSS file is only for one document, don't use external code; instead, embed it in the page.
- Use inline images whenever you can.
- Use CSS sprites.
- Reduce the use of images for effects, titles, and text. Try to meet all of these needs using only CSS.
- Use multipart documents when compatible.
- Download only the initially required code and resources and then, after the on load event, download all the rest on Ajax devices (lazy loading).



Every mobile browser supports a cache for resources, and you should definitely use it, with a long-lived expiry for each static resource. Analyze how the cache works (this is outside the scope of this book) and make it your friend, not your enemy!

Compressing

Compression is a necessity, and there are different techniques you can use for it:

- Minimize your HTML files, removing spaces, comments, and nonuseful tags.
- Minimize your CSS files, removing spaces and comments.
- Minimize your JavaScript files, removing spaces and comments and obfuscating the code.
- Use HTTP 1.1 compression for delivering static and dynamic text-based files (HTML, JavaScript, CSS, XML, JSON).
- Use a cookie-free domain (or alias domain) for static content files.

There are plenty of online and offline tools for minimizing files, such as JSMIn (<http://crockford.com/javascript/jsmin>) and YUI! Compressor (<http://developer.yahoo.com/yui/compressor>).

Going Beyond JavaScript Compression

There are plenty of good JavaScript obfuscators and minimizing tools out there, but Google has taken an extra step and created the *Closure Compiler*, a new concept in JavaScript programming. It is not just a minimizing and obfuscating tool, but it is also a compiler: it compiles JavaScript code into better JavaScript code and is very helpful for mobile websites.

You can download the compiler at code.google.com/closure/compiler or use the web application compiler at closure-compiler.appspot.com.

The code will be rewritten to be lighter and quicker to execute. The resulting code will not be suitable for human reading because it will not use good programming practices, but that is not the goal. We are not going to edit the resulting code; we will always work with the original code (with comments and all the best practices) and recompile it before sending it to the server.

HTTP compression

HTTP 1.1 added compression (using GZIP and *deflate*) as an optional possibility when delivering a file to the client. Using this option is strongly recommended for text-based files on most mobile devices, because it will reduce the traffic between the server and the client by up to 80%. It will add some overhead on the client (to uncompress the content), but it's well worth it. Network traffic will be one of our worst problems if the user is not connected to a WiFi network. Even with 3G connections, the network can have latency problems.



If you work with ASP.NET Web Forms, you should be careful about the usage of the ViewState, which generates big hidden input tags in the HTML. Deactivate the ViewState on controls where you won't use it.

You will find plenty of resources on the Web about how to configure HTTP compression for Apache, Internet Information Server, and other products. The most important thing you need to remember is to also compress dynamic scripts delivering markup, like PHP scripts, which by default do not use HTTP compression in Apache.



As covered in Chapter 9, some mobile browsers already support the SPDY protocol, which improves performance. Implement it when you have users browsing your website on compatible devices.

Other tips

Here are some other tips to keep in mind:

- Compress images and choose the best format and color palette. You can use the free online tool Smush.it (<http://www.smushit.com>) from Yahoo!.
- Deliver small images for small screens. You can use a dynamic resizing tool, or the free online services covered in Chapter 10.
- Reduce the initial load time as much as possible. You want the web application to be ready as soon as possible.
- Minimize DOM access and simplify your document structure.
- Use HTML5 storage for caching data and resources in base64. You can place a cookie on the client, so the next time the user accesses the website it will know that resources are already stored on the client and don't need to be injected at that time.
- Flush the buffer early, using `flush()` in PHP or `Response.Flush()` in ASP.NET, after `</header>` and after big blocks of visual components.
- Avoid redirects between pages, especially in the home page.
- Use touch events to improve responsiveness, as covered in Chapter 15.
- In very large documents and web apps, using iframes to split the DOM into smaller parts may improve performance.
- Create DOM object pools for reuse instead of creating lots of new elements that will remain in memory when they are off the screen or not useful anymore.
- Create a quick and simple home page.
- Put `script` tags at the bottom to avoid resource download delays and/or use the new HTML5 script attributes covered in Chapter 12, such as `defer`.

- Use a content delivery network or a static server for static content if you have a lot of images or other static content.
- Remove any nonuseful headers from the server responses (like server identification or “powered by”).
- Some CSS3 features (mostly on large elements) can impact performance, such as shadows and gradients. Use measurement tools to find problem areas in your project.

Deferred JavaScript Evaluation

The Gmail team, in conjunction with Charles Jolley (<http://blog.sproutcore.com>), has created a very clever and simple way of reducing the initial payload of JavaScript execution. The solution is to deliver the JavaScript code inside a comment block (`/* */`). This means the JavaScript isn’t executing while the page is loading, and it doesn’t freeze the UI or block other resources.

When you need to execute that library or code, you just get the script by ID, get its content, remove the comment characters, and just `eval()` the code. Pretty smart, isn’t it? On an iPhone running iOS 2.2, 200 KB of JavaScript code adds 2.6 seconds to the initial page load time, while if it is commented it adds just 240 ms. After all the initial loading is done (or later, whenever you need it), you can parse the code.

JavaScript performance

Again, keep your code simple. Here are some other specific tips for mobile JavaScript coding:

- Don’t use try/catch expressions for expensive code.
- Avoid using `eval()`, even in situations where you might not think about it being used, like when using a string in `setTimeout()` instead of a function.
- Avoid using `with`.
- Minimize the usage of global variables.
- Avoid the usage of JavaScript frameworks that are not used totally.
- Minimize the number of changes in the DOM, and make the changes in the same operation. Many browsers repaint the whole screen on each change.
- Implement a timeout for Ajax calls.
- Compress (and if you want, compile with Closure Compiler) your code.
- Use CSS transitions/animations instead of JavaScript-based animations.

- In compatible browsers, use the Animation Timing API (also known as `requestAnimationFrame`) instead of `setInterval()`.



For a while there was a performance tip advocating the use of CSS 3D transforms (for example, `translateZ` or `translate3d`) instead of 2D transforms, as only the former were hardware-accelerated. On modern browsers (from 2012) that is no longer the case, and it can even harm performance on the video card. Modern browsers are smart enough to hardware-accelerate what they can to improve performance.

An excellent book about JavaScript performance tips is Nicholas Zakas's *High Performance JavaScript*, also from O'Reilly.

Distribution and Social Web 2.0

So, you've finished your mobile web application and you are ready to go to market (or so you think). However, your work isn't over yet. In this chapter, we are going to analyze some search engine optimization tips, and talk about how to get users to actually visit our mobile websites and encourage them to come back again and again.

We will also explore how to monetize mobile websites using advertising and how to merge our applications with some social features, such as SMS, RSS, and social networks.

Mobile SEO

Search engine optimization (SEO) refers to a set of best practices that you can follow to help your website be in the best possible place in a search engine's results.

In general, typical desktop SEO techniques apply to mobile websites, too, but some extra care must be taken. As we've already discussed, generating too much code (metadata) and too much text for keyword crawling is not the best solution for the mobile web.

The first thing we need to understand is that mobile search users are not the same as desktop search users. Mobile users are typically searching for something very specific, and we should do our best to facilitate access to those resources.

Mobile search engines (Google, Yahoo!, Bing) localize the search results, so if your service is location-based, you should make sure that your location is properly defined in your text and code. In mobile search engines, the user only types a few characters and the engine tries to suggest the best possible results based on location and previous results, with mobile-specific content given priority.



If your mobile website gets content using Ajax, you should implement Google's proposal for making the content being indexed crawlable. You can find more information about this at code.google.com/web/ajax-crawling.

Search engines like Google will try to serve mobile-specific content first, but if someone is looking for the exact name of your application and Google doesn't know that you have a mobile website, the user will be redirected to your desktop site or to a transcoded mobile version of it produced by a Google server.

If you appear in the search engine's databases, you will also be found using the native applications that many search engines are developing, including voice-powered search applications.

Spiders and Discoverability

The first problem is how to make your mobile website known to the search engines. This can be different depending upon whether you already have a desktop website that has been crawled or not.

If you already have a desktop website, you can give search engines the URL of your mobile site using the alternate link method:

```
<link rel="alternate" media="handheld" href="http://m.yoursite.com" />
```

You can also add your mobile site manually, using these links:

- Bing (<http://bing.com/webmaster>)
- Google (<http://google.com/addurl>)

Mobile Sitemaps

Google has created an extension to the Sitemap protocol (<http://sitemaps.org>) for mobile web content discoverability, called Mobile Sitemaps. After creating an account in Google Webmaster Central (<http://www.google.com/webmasters>), you can add your mobile site to Google's database. You will need to verify that you are the owner of the site, by inserting a temporal meta tag or HTML file in your site.



Googlebot-Mobile uses the Accept HTTP header to determine whether a site delivers mobile content types. If you want to be sure that the bot can access your site, you can also check that the User-Agent header contains Googlebot-Mobile. Some sites will only allow access to mobile devices, and while Googlebot-Mobile tries to emulate such a device it is not always successful in gaining access unless it is specifically allowed.

Once your site has been validated, you can submit a Sitemap for it. If your mobile site is targeted to only one country using a noncountry top-level domain (like *.com* or *.mobi*), you can also define the geographic target for which your mobile site is prepared.



Check the Sitemaps documentation (<http://sitemaps.org>) for full tag and option support.

A Mobile Sitemap is an XML file, based on the Sitemap standard, that lists the mobile URLs for your site (XHTML, XHTML MP, WML, cHTML). You can provide URLs for both mobile and nonmobile versions depending on the headers, but you should not list non-mobile-only URLs. A sample Sitemap file looks like this:

```
<?xml version="1.0" encoding="UTF-8" ?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9"
  xmlns:mobile="http://www.google.com/schemas/sitemap-mobile/1.0">
  <url>
    <loc>http://m.yourdomain.com/</loc>
    <mobile:mobile/>
  </url>
</urlset>
```

You should provide one url element for each mobile URL and page, including the *mobile:mobile* empty tag. If you have many versions using different URLs (for iPhone, WML, etc.), you should provide them all in the same file.



To check whether your website is listed in Google Mobile, visit *m.google.com* with a mobile device and use the search operator *site:your_domain*.

The Google Webmaster Team also suggests detecting Googlebot-Mobile in your desktop site and redirecting it to the mobile-specific version of the same page. For example, for information about a product X in your desktop site, you should redirect the bot to the

mobile URL displaying information about that product. Otherwise, Google will use a transcoder on the desktop page.

How Users Find You

Search engines are not the only way for users to discover your mobile website. Obviously, offline marketing is always welcome, but there are also other online features we should implement to facilitate discoverability. These include advertising the new mobile website to your current desktop visitors and implementing newsletters and feed readers.

The first problem to tackle is simplifying the user's first access to the mobile website. Many mobile users still don't know how to go to a URL if it is not on the carrier's home page, and many others will not want to type a long URL on a numeric keypad device.

SMS Invitation

A good solution is to include in your desktop website a form to collect the user's phone number and then send him a WAP Push or an SMS link. A WAP Push is a special message with a URL inside. This is generally a premium SMS, and some carriers don't allow sending them from a website.

An SMS link is just a normal SMS with a link inside. Almost every modern device with a browser will autodetect a URL inside a text message if it begins with *www* or *http://* and will convert the URL into a link that the user can click after receiving the SMS.

The big question is, how do we send an SMS from a website? The answer is not what you might expect—there isn't a simple or free way to do it. We have to use an SMS provider or gateway that, with a simple web service call, will send the message to users in one country or worldwide. We will have to pay for that SMS, but depending on the business, a new mobile web user will probably be worth the small expense.



Some SMS gateway providers also allow inbound SMS messages that will be routed to your scripts or will be accessible via an API. This could be an excellent solution to receive queries by SMS to your service.

Some SMS gateway providers include:

- Mogreet (<http://developer.mogreet.com>)
- Lleida (<http://www.lleida.net>)
- Clickatell SMS Gateway (<http://www.clickatell.com>)
- BulkSMS (<http://www.bulksms.com>)
- ClockWork (<http://www.clockworksms.com>)

- Twilio SMS (<http://www.twilio.com/sms>)
- SendHub (<http://www.sendhub.com/developer>)

Alternatively, you can install a 3G or GPRS modem on your server or in any machine and develop a little SMS gateway of your own, with a corporate or personal account. A widget or an application on your device could also work, although this is not the preferred way.



Most carriers support SMS sending and receiving through an API, but you'll have to sign up with and pay each carrier separately.

You can also use carrier developer networks and the up-and-coming OneAPI to send messages to known networks.

Email Invitation

If the user accesses her email mailbox from her mobile device, an alternative to SMS is to send the user a free email message containing the mobile URL.

Mobile Tiny URL

To enable the user to type your URL easily, you can use the free service Mobile Tiny URL (<http://www.mobiletinyurl.com>). It converts any URL into a short form that can be typed with only 13 keypresses on a numeric or QWERTY keypad. By default the generated short URL doesn't work in desktop browsers, but you can add desktop support. These short URLs are useful for publication on desktop websites and in printed advertising.

For example, instead of typing *m.safaribooksonline.com*, a mobile user can type *ad.ag/admtgp* (saving 57 keypresses). As you can see, the generated URL uses only the first letters associated with every key on a numeric keypad, to speed up entry. Even apparently simple URLs like *google.com* will require 37 keypresses on a mobile phone's numeric keypad, and you can save 24 by using the compressed URL (*ad.ag/tgtmjg*). For smartphones and tablets with full QWERTY keyboards, if you have a short domain you can just advertise it; Mobile Tiny URLs are more useful on feature and social phones.

QR Codes

A QR code is a two-dimensional barcode (also called a *matrix code*) that allows the storage of several bytes in a graphic. These codes have many uses, one of which is to provide a URL that can be read by devices with bar code readers. Many Nokia and

Android devices come with these readers preinstalled, but on other devices, users will need to download one. A sample QR code is shown in Figure 19-1.



Figure 19-1. Google Maps created the Google Places campaign, sending businesses (like restaurants) stickers like this one that users can scan with their mobile devices to access information, reviews, etc.



A QR code can contain 4,296 alphanumeric characters, or 2,953 bytes for binary data. Some devices also support other data inside, like contact information (for example, a vCard file).

QR codes are well known in mobile advertising; many campaigns use these codes in newspapers, on street signs, and even on t-shirts.

To create a QR code, you can use any of these free services:

- Kaywa (<http://qrcode.kaywa.com>)
- Create QR Code (<http://createqrcode.appspot.com>)
- Mobile Tiny URL (<http://www.mobiletinyurl.com>)

If you need to generate a QR code dynamically, there are libraries for almost all server-side platforms that will generate the right image for you.



Google's URL Shortener (<http://goo.gl>) allows us to create a QR code by adding a .qr suffix to the short URL; for example, <http://goo.gl/C9pYl.qr>.

NFC Tags

Modern smartphones and tablets include the Near Field Communication (NFC) technology, allowing the device to “touch” a tag and read some information from it. A tag can be embedded in a product, a sign, a box, or anywhere.

An NFC tag can store a URL, whose length depends on the tag type: a standard Ultralight NFC tag can store up to 41 characters, while other tags can store up to 132 characters.

To create an NFC tag, you can either buy them online from some providers (you provide the URL and the tag type you want and they will create them and send them by postal mail), or you can buy blank tags and write them yourself using an Android device. If you want to learn more about how to create your own tags, check out the article (<http://bit.ly/16uYVTZ>).



Tagstand Writer (<http://writer.tagstand.com>) is an Android app that will let you write URLs on blank NFC tags that you need to buy. The website provides a list of sites where you can buy tags.

User Fidelizing

Once you've gotten a user to your mobile website, how do you encourage him to come back and maintain an interest in your service? If you are not providing a must-use service (such as online banking or email), you may want to implement some of the following techniques to “fidelize” your users:

- Encourage the user to add your site to her bookmarks, or to the Home screen on selected platforms.
- Offer the user an app with richer features.
- Offer the user a shortcut to download for the applications menu or Home screen.
- Offer the user a home widget on supported devices, with automatic updates.
- Create a mobile RSS news feed.
- Provide an SMS alert subscription.

Web Shortcuts

A *web shortcut* is a native application or web app that has an icon in the applications menu that launches the browser when it is activated. Adding a shortcut is better than adding a bookmark, because it will be installed just like any other application.

RSS

Some browsers (Opera, Bolt, NetFront, Symbian) detect feed meta tags and offer the user the option to subscribe to the feeds to get updates on the sites that provide them. To offer this service, you should provide an RSS file with a mobile web link inside:

```
<link rel="alternate" type="application/rss+xml" title="Mobile RSS"
      href="http://mobilexweb.com/rss.xml" />
```

If you have an RSS file for your mobile content, there are plenty of online resources that will create a full native app for different platforms based on your RSS content.



Nokia offers the free Xpress Web App Builder (<http://xpresswebapps.nokia.com>) for Series 40 that will take your website's content, including RSS, and create a fully capable web app.

Open Search

If your website provides a search engine, you should supply an Open Search description document that will allow users of compatible devices to add your engine to the list of possible search engines. Not too many mobile browsers support this format at the time of this writing, but a mobile extension is in draft (see Open Search (<http://opensearch.org>)).

To define an Open Search declaration for discoverability, use the following link tag:

```
<link rel="search" type="application/opensearchdescription+xml"
      href="http://mobilexweb.com/opensearch.xml" />
```

The Open Search descriptor file will look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<OpenSearchDescription xmlns="http://a9.com/-/spec/opensearch/1.1/">
  <ShortName>Mobile Web Search</ShortName>
  <Description>Search in our mobile web</Description>
  <Url type="text/html"
        template="http://mysite.com/?q={searchTerms}" />
  <Image height="64" width="64" type="image/png">
    http://example.com/icon.png
  </Image>
  <Language>en-us</Language>
</OpenSearchDescription>
```

Apple Passbook

Passbook is a native app available in iOS from 6.0 that works as a virtual container for all your passes, tickets, discount coupons, loyalty cards, and gift cards. As a web developer you may want to serve the user with a discount coupon, a ticket to an event, an e-ticket for his next flight, or a loyalty card.

Apple allows websites to deliver these kinds of passes from a website without the need of a native app, as you can see in Figure 19-2.

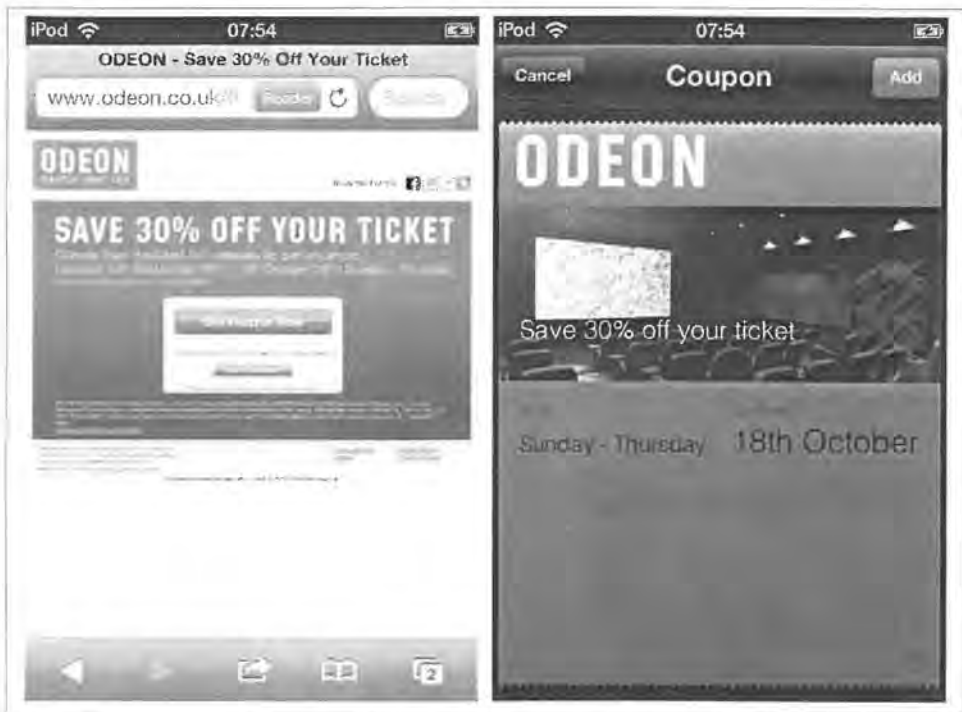


Figure 19-2. Odeon Cinemas (<http://odeon.co.uk>) in UK offers discount codes to iOS Passbook directly from their website.

To deliver the pass via your website you just need to use the MIME type application/vnd.apple.pkpass or send the file as an email attachment.



Apple provides a tool that you can install on your server to package and sign customized passes on the fly that may include current user information.

A Passbook file is a package including just a JSON metadata file and a couple of images. You need to package the file and sign it. Unfortunately, to sign the pass you need a signature from Apple, and that means you need an iOS Developer Program account (\$99/year). If you receive the pass already signed, you can just insert it on your own site.

One of the great features of passes is that once they're installed you can provide some web services on your end, and (through Apple's Push Notification Service) the operating system will call your web services to update the information on the passes. Therefore, when you deliver a pass to the user, you can update it from your end without a native app.

More information on Apple's Passbook is available at Apple's Developer website (<http://developer.apple.com/passbook>).

Mobile Web Statistics

With your website online, you will want to gather some statistics about usage, visitors, and even mobile browsers accessing your site. Typical desktop web statistics systems don't work well with mobile sites, because they don't have mobile user agents in mind when analyzing logs and don't work on non-JavaScript devices or other kinds of services.

However, there are some free and commercial solutions available for mobile web statistics.

Google Analytics for Mobile

Google Analytics is one of the most used (and most powerful) free web statistics tools for mobile websites, but it is based on JavaScript code that, in the mobile world, only works on high-end devices.

Google Analytics for Mobile Websites (<http://bit.ly/Wgbfob>) works on all web-enabled browsers, with or without JavaScript support. The technology supports script code for PHP, ASP.NET, Perl, and JSP.



Google Analytics is also available for Android and iPhone native applications, supporting tracking of pages and events.

To use the mobile service, you should apply for a normal Google Analytics (<http://www.google.com/analytics>) account. Create a new Website Profile and, in the Advanced section, select "A site built for a mobile phone," select your server language, and follow the instructions.

Yahoo! Web Analytics

Yahoo! Web Analytics (<http://web.analytics.yahoo.com>) supports mobile devices, giving the following statistics about your mobile users:

- Mobile device manufacturer
- Mobile device model
- Model device screen size
- Carrier name

You will find these statistics in the Mobile Reports section.

Mobilytics

Mobilytics (<http://www.mobilytics.net>) provides free and premium metric and visual indicators about your mobile web visitors. It offers scripts for PHP, ASP.NET, JSP, and other server-side platforms. Transcoder detection, mobile device detection, and other capabilities are available in premium plans.

Monetizing Your Website

You've put a lot of work into building your site. How can you make a return on that investment? One solution could be advertising. Another solution is to charge for content using Premium SMS (through carriers) or Facebook Payments (<http://developers.facebook.com/docs/payments>), which is compatible with mobile websites.

Mobile Advertisements

If you have a free mobile website and you want to monetize it with advertisements, there are a few solutions that you can use. Today, most ad networks offer a mobile version of their services.



For more information about mobile advertising, visit the Mobile Marketing Association (<http://www.mmaglobal.com>).

Google AdSense for mobile content

With Google AdSense, you can easily insert mobile-optimized ads for mobile devices. To use it, log in or sign up at Google AdSense (<http://www.google.com/adsense>) and select "AdSense for Mobile Content" on the Account Setup tab.



When you define a mobile campaign for Google AdWords, you can use the Keyword tool to access information about mobile searches in the Google ecosystem.

It is possible to select only advertisements appropriate for the iPhone and other high-end devices, or for all devices (including ads in XHTML, cHTML, and WML format), using server-side code. When you add the server code (for example, PHP) you should start seeing mobile advertisements on your site within 48 hours.



The Interactive Advertising Bureau (IAB) has released a document with the suggested formats and sizes for mobile ads and banners. It can be found at www.mobilexweb.com/go/mobileads.

Mobile Web Social Features

Your mobile website will not be complete if you don't add some social features to it. In the current social networking era, social integration is a must-have feature to implement.

Authentication and Sharing APIs

Facebook, Twitter, and Google all offer APIs that can enhance the visibility of your mobile site thanks to the sharing features.

Facebook Connect

Facebook offers Facebook Connect for Mobile Web (<http://developers.facebook.com/mobile>), an HTML5 library that lets our applications log users in using their Facebook accounts. With this API you can:

- Create a login mechanism easily.
- Get user session data.
- Call methods from the Graph API and prompt for extended permissions (access friends list, send private messages).
- Post on the Facebook timeline.

To use it, you will need to register an app on Facebook, implement the Facebook SDK, log the user in, and use the Graph API.

To get detailed information and samples, visit Facebook developer's site (<https://developers.facebook.com/docs/guides/mobile/web>).

Twitter for Websites

Twitter for Websites (<http://dev.twitter.com>) allows us to integrate our mobile website with Twitter login. We can also access the user's account, such as for posting new messages or accessing the whole Twitter API.

Google APIs

Google offers different APIs that are useful for mobile websites, including Google Accounts Authentication and Authorization (<https://developers.google.com/accounts>), useful to log in the user, and the Google+ API (<https://developers.google.com/+api>) for sharing on that social network.

Sharing Content

For any content you are serving in your mobile website, you can offer a *Share* service to publish the URL via Twitter, Facebook, and other social networks.



Remember from Chapter 7 that we have to use some specific meta tags defining the information to share so the sharing mechanism works better. Remember also that some browsers, such as Safari on iOS and Internet Explorer 10 on Windows 8, have a sharing ability built in.

For most social networks, you should use the same URL you would use for the desktop website. On the server, the social network scripts will redirect mobile users to the mobile website.

For Twitter, you can use a link like this:

```
http://twitter.com/home?status=<your message here>
```

Remember that Twitter has a limit of 140 characters, including an optional URL using *http://*, which should be URL-encoded in the status variable. For long URLs, you should use a shortener service API.

For Facebook, you can share a link using:

```
http://m.facebook.com/sharer.php?u=<url to share>&t=<title of content>
```


MIME Types for Mobile Content

Markup and Script MIME Types

Table A-1. Markup and script MIME types

Format	Typical extension	MIME type
WML	.wml	text/vnd.wap.xml
WMLScript	.wmls	text/vnd.wap.wmlscript
HTML 3/4/5	.html	text/html
cHTML	.html	text/html
XHTML	.html, .xhtml	application/xhtml+xml, text/xml, text/html
XHTML MP	.html, .xhtml	application/vnd.wap.xhtml+xml, application/xhtml+xml, text/xml, text/html
JavaScript	.js	text/javascript, application/ecmascript, application/javascript
CSS2, CSS3, WAP CSS, and CSS MP	.css	text/css
Multipart document		multipart/mixed, application/vnd.wap.multipart.mixed

Image MIME Types

Table A-2. Image MIME types

Format	Typical extension	MIME type
GIF	.gif	image/gif
JPEG	.jpeg, .jpg	image/jpeg, image/jpg
PNG	.png	image/png
SVG	.svg	image/svg+xml

Format	Typical extension	MIME type
WEBP	<i>.webp</i>	image/webp
Compressed SVG	<i>.svgz</i>	image/svg+xml
WBMP	<i>.wbmp</i>	image/vnd.wap.wbmp
Nokia Wallpaper		image/vnd.nok-wallpaper

Mobile Content MIME Types

Table A-3. Mobile content MIME types

Format	Typical extension	MIME type
iOS Passbook	<i>.pkpass</i>	application/vnd.apple.pkpass
Java ME Application Descriptor	<i>.jad</i>	text/vnd.sun.j2me.app-descriptor
Java ME Archive	<i>.jar</i>	application/java-archive
BlackBerry Archive	<i>.cod</i>	application/vnd.rim.cod
Android Application	<i>.apk</i>	application/vnd.android.package-archive
Symbian Application	<i>.sis</i>	application/vnd.symbian.install
Symbian Application	<i>.sisx</i>	x-epoc/x-sisx-app
vCalendar	<i>.vcalendar</i>	
iCalendar	<i>.icalendar</i>	
Nokia Flash Format	<i>.nfl</i>	
Adobe Flash Movie	<i>.swf</i>	application/x-shockwave-flash
XML	<i>.xml</i>	text/xml
JSON	<i>.json</i>	application/json, text/json, text/javascript
RSS	<i>.rss, .xml</i>	application/rss+xml
Open Search Description	<i>.xml</i>	application/opensearchdescription+xml
Mobile Sitemap	<i>.xml</i>	text/xml
Multimedia Message	<i>.mms, .smil</i>	application/vnd.wap.mms-message
Bookmark		application/x-wap-prov.browser-bookmarks
Sony Ericsson MMS Template	<i>.tpl</i>	application/vnd.sonyericsson.mms-template
OMA Download		application/vnd.oma.dd+xml

Audio and Video MIME Types

Table A-4. Audio and video MIME types

Format	Typical extension	MIME type
3GPP	<i>.3gp, .3gpp</i>	video/3gpp
3GPP 2	<i>.3gp2, .3gpp2</i>	video/3gpp2
QuickTime MOV	<i>.mov</i>	video/quicktime
Windows Media Video	<i>.wmv</i>	video/x-ms-wmv
Windows Media Audio	<i>.wma</i>	audio/x-ms-wma
Real Video	<i>.rv</i>	video/vnd.rn-realvideo
Real Audio	<i>.ra, .ram</i>	audio/x-pn-realaudio
MP3	<i>.mp3</i>	audio/mp3
Flash Video	<i>.flv</i>	video/x-flv
MPEG-4	<i>.mp4</i>	video/mp4
Audio WebM	<i>.webm</i>	audio/webm
Video WebM	<i>.webm</i>	video/webm

Webapp MIME Types

Table A-5. Webapp MIME types

Format	Typical extension	MIME type
Symbian WRT webapp	<i>.wgz</i>	application/x-nokia-widget
Samsung widget	<i>.wgt</i>	application/vnd.samsung.widget
JIL widget	<i>.wgt</i>	application/widget
Opera widgets	<i>.wgt</i>	application/x-opera-widgets
HTML5 Manifest file	<i>.manifest</i>	text/cache-manifest
Server-Sent Event Stream		text/event-stream

Symbols

- 2D context, 357
- 3D image formats, 333
- 3D screens, 53
- 3GP container, 369
- 3GP2 container, 369
- 3GPP (3rd Generation Partnership Project), 351
- 51 Degrees website, 61, 328
- 80/20 law, 140

A

- a element, 553
- A-GPS (Assisted GPS), 60, 530
- AAC codec, 368
- absolute positions, 405
- Accelerometer API, 583, 594, 649
- Accelerometer object (Cordova/PhoneGap)
 - clearAcceleration method, 583
 - getCurrentAcceleration method, 583
 - watchAcceleration method, 583
- accelerometers, 60, 60, 579
- Accept request header (HTTP), 301, 696
- Accept-Charset request header (HTTP), 301
- Accept-Language request header (HTTP), 301
- accessibility (HTML5), 232
- accesskey attribute, 230, 236
- accuracy in location technology, 529
- active pseudoclass (CSS), 413
- Add to Home Screen library, 611
- addEventListener() method
 - cache events and, 508
 - managing events, 477
 - transition ending and, 441
- Adobe Creative Cloud services, 115
- Adobe Device Central, 111
- Adobe Dreamweaver, 89, 157
- Adobe Edge Animate, 90, 442
- Adobe Edge Code, 90
- Adobe Edge Inspect, 90, 115–116
- Adobe Edge Reflow, 90
- Adobe Flash Media Serve, 370
- Adobe Flash Player, 123, 365–366
- Adobe HTTP Dynamic Streaming, 376
- Adobe Intel
 - native web apps for, 91
- Adobe PhoneGap, 72
 - (see also Apache Cordova framework)
 - about, 72
 - native web apps for, 644–650
- advertisements on mobile websites, 705
- Advertising API, 627
- after pseudoclass (CSS), 415
- Ajax
 - about, 522–523
 - cross-domain requests, 523
 - JSONP and, 524
 - with offline resources, 525

We'd like to hear your suggestions for improving our indexes. Send email to index@oreilly.com.

- ajax capability group (WURFL), 316
- ajax_supports_javascript capability (WURFL), 318
- ajax_support_event_listener capability (WURFL), 318
- Akamai service provider
 - Internet Observatory website, 61
 - Mobitest tool, 687
 - streaming services for mobile devices, 377
- Alarm API, 595
- alert dialog (JavaScript), 467
- alert windows, 684
- alt attribute, 336
- Amazon AppStore, 14, 651
- Amazon Silk browser, 76, 311
- Amazon.com, 28
- Ambient Light Events API, 594
- AND operator, 386
- Android platform
 - about, 13
 - apple-touch icon and, 193–195
 - application and games delivery, 665
 - brands and models in, 13–16
 - Debug Bridge tool, 682
 - debugging tools, 681
 - distributing native web apps, 651
 - emulator for, 94–98
 - fragmentation and, 16
 - Google Maps application, 541
 - intents, 562
 - online site on screen sizes, 54
 - preinstalled browser, 74
 - versions, 14
- Android SDK Manager, 94
- Android Virtual Device (AVD) Manager, 95
- animated GIFs, 332, 336
- Animated PNG (APNG), 333, 336
- animation property (CSS), 442
- animation timing API, 475, 693
- animation-delay property (CSS), 442
- animation-direction property (CSS), 442
- animation-duration property (CSS), 442
- animation-fillmode property (CSS), 443, 444
- animation-iteration-count property (CSS), 442
- animation-name property (CSS), 442
- animation-playstate property (CSS), 443
- animation-timing-function property (CSS), 442
- animations
 - about, 442–443
 - keyframes at-rule, 443–445
 - listening for events, 446
 - timing control for, 475
 - transitions and, 439–441
- Apache Cordova framework
 - about, 72
 - APIs for mobile web apps, 594, 649
 - managing cameras, 592
 - native web apps for, 70, 91, 644–650
 - sensor support, 583
 - Weinre tool, 683
- Apache DeviceMap, 328
- Apache platform, MIME types and, 655
- apk files, 665
- APNG (Animated PNG), 333, 336
- APNG-canvas polyfill, 333, 336
- app stores, 14
 - (see also specific stores)
 - Android devices, 14
 - Blackberry devices, 25
 - distributing native web apps, 650
 - Nokia devices, 20
 - Samsung devices, 26
- AppBar control, 617
- AppBarCommand control, 617
- Appcelerator Titanium framework, 72, 91
- appearance property (CSS3), 421
- Apple Corporation
 - about, 9
 - Adobe versus, 366
 - mobile devices, 9–13
- Apple Dashboard Widget, 602
- Apple Gesture API, 574–576
- Apple HTTP Live Streaming (HLS), 376–377
- Apple Passbook, 703
- Apple Touch Events API, 565–568
- Apple TV, 375
- apple-mobile-web-app-title meta tag, 191
- apple-touch icon, 187–195
- application and games delivery
 - Android applications, 665
 - iOS applications, 663–664
 - Java ME and, 667–669
 - Windows applications, 666
- Application API, 627
- application cache
 - about, 501
 - compatibility and limits, 508–511
 - fallback mechanism, 504

- manifest file and, 502
- uses for, 502
- applicationCache object
 - about, 506–506
 - addEventListener() method, 508
 - cached event, 507
 - checking event, 507
 - downloading event, 507
 - error event, 507
 - events available to, 507
 - noupdate event, 507
 - obsolete event, 507
 - progress event, 507
 - status property, 506
 - swapCache() method, 506
 - update() method, 504, 506, 506
 - updateready event, 507
- AppManager API, 640
- Apps API (Mozilla Open Web apps), 622
- AppThwack service, 119
- Aptana tool, 91
- AR (Augmented Reality), 530
- arc() method (Canvas API), 358
- architecture and design
 - mobile strategy for, 127–140
 - navigation tips, 140
 - user experience, 141–152
- ArrayBuffer data type, 464
- ASP.NET
 - MIME types and, 657
 - reading headers, 308
- aspect-ratio attribute, 386
- Assisted GPS (A-GPS), 60, 530
- async attribute, 453
- Asynchronous JavaScript and XML (see Ajax)
- AT&T Location API, 539
- audio
 - about, 379
 - compatibility table for web platforms, 381
 - containers and codecs, 368–369
 - invisible player, 380
 - MIME types supported, 711
 - streaming, 376–377
 - Web Audio API, 380
- AudioContext constructor, 380
- Augmented Reality (AR), 530
- authenticating connections, 706
- autocomplete attribute, 267, 499
- autofocus attribute, 267, 297

- autogrowing textarea, 496
- automatic links, removing, 216
- autoplay attribute, 374
- autosave attribute, 245
- AVC (H.264 codec), 368
- AVD (Android Virtual Device) Manager, 95
- AVI container, 368

B

- backbutton event (Cordova/PhoneGap), 648
- backface-visibility property (CSS), 436
- background execution
 - about, 456–457
 - background tab notification trick, 460
 - compatibility table, 461
 - push notifications, 462
 - status detection, 457–460
- background tab notification trick, 460
- background-attachment property (CSS), 414
- background-clip property (CSS3), 414
- background-image property (CSS), 285, 345, 413
- background-origin property (CSS3), 414
- background-position property (CSS), 338, 422
- background-repeat property (CSS), 413
- background-size property (CSS3)
 - about, 285, 338, 414
 - percentage values in, 349
- Bada emulator, 111
- Baidu browser, 86
- Baidu Yi operating system, 31
- bandwidth simulators, 674
- base64 conversion, 334
- baseJS library, 486
- battery consumption, 455, 593
- Battery Status API
 - about, 593
 - charging attribute, 593
 - chargingTime attribute, 593
 - dischargingTime attribute, 593
 - level attribute, 593
- batterycritical event (Cordova/PhoneGap), 648
- batterylow event (Cordova/PhoneGap), 648
- batterystatus event (Cordova/PhoneGap), 648
- BB10 simulator, 107
- BBM API, 627
- bbUL.js framework, 488, 628–629
- bearer capability group (WURFL), 316
- before pseudoclass (CSS), 415

- manifest file and, 502
- uses for, 502
- applicationCache object
 - about, 506–506
 - addEventListener() method, 508
 - cached event, 507
 - checking event, 507
 - downloading event, 507
 - error event, 507
 - events available to, 507
 - noupdate event, 507
 - obsolete event, 507
 - progress event, 507
 - status property, 506
 - swapCache() method, 506
 - update() method, 504, 506, 506
 - updateready event, 507
- AppManager API, 640
- Apps API (Mozilla Open Web apps), 622
- AppThwack service, 119
- Aptana tool, 91
- AR (Augmented Reality), 530
- arc() method (Canvas API), 358
- architecture and design
 - mobile strategy for, 127–140
 - navigation tips, 140
 - user experience, 141–152
- ArrayBuffer data type, 464
- ASP.NET
 - MIME types and, 657
 - reading headers, 308
- aspect-ratio attribute, 386
- Assisted GPS (A-GPS), 60, 530
- async attribute, 453
- Asynchronous JavaScript and XML (see Ajax)
- AT&T Location API, 539
- audio
 - about, 379
 - compatibility table for web platforms, 381
 - containers and codecs, 368–369
 - invisible player, 380
 - MIME types supported, 711
 - streaming, 376–377
 - Web Audio API, 380
- AudioContext constructor, 380
- Augmented Reality (AR), 530
- authenticating connections, 706
- autocomplete attribute, 267, 499
- autofocus attribute, 267, 297

- autogrowing textarea, 496
- automatic links, removing, 216
- autoplay attribute, 374
- autosave attribute, 245
- AVC (H.264 codec), 368
- AVD (Android Virtual Device) Manager, 95
- AVI container, 368

B

- backbutton event (Cordova/PhoneGap), 648
- backface-visibility property (CSS), 436
- background execution
 - about, 456–457
 - background tab notification trick, 460
 - compatibility table, 461
 - push notifications, 462
 - status detection, 457–460
- background tab notification trick, 460
- background-attachment property (CSS), 414
- background-clip property (CSS3), 414
- background-image property (CSS), 285, 345, 413
- background-origin property (CSS3), 414
- background-position property (CSS), 338, 422
- background-repeat property (CSS), 413
- background-size property (CSS3)
 - about, 285, 338, 414
 - percentage values in, 349
- Bada emulator, 111
- Baidu browser, 86
- Baidu Yi operating system, 31
- bandwidth simulators, 674
- base64 conversion, 334
- baseJS library, 486
- battery consumption, 455, 593
- Battery Status API
 - about, 593
 - charging attribute, 593
 - chargingTime attribute, 593
 - dischargingTime attribute, 593
 - level attribute, 593
- batterycritical event (Cordova/PhoneGap), 648
- batterylow event (Cordova/PhoneGap), 648
- batterystatus event (Cordova/PhoneGap), 648
- BB10 simulator, 107
- BBM API, 627
- bbUI.js framework, 488, 628–629
- bearer capability group (WURFL), 316
- before pseudoclass (CSS), 415

- beforeunload event, 477
- beginPath() method (Canvas API), 358
- bezierCurveTo() method (Canvas API), 358
- binary data, 464
- Bing Maps application, 545
- Bing search engine, 695
- bingmaps: protocol, 545
- BlackBerry 10 platform, 26, 78
- BlackBerry devices
 - about, 24
 - apple-touch icon and, 192
 - preinstalled browser, 78
 - Remote Web Inspector, 682
 - simulators for, 104–107
 - touch behavior, 568
 - TunnelTilt game, 363
 - viewport declaration and, 214
 - WebWorks support, 624–632
- BlackBerry Email and MDS Services Simulator Package, 105
- BlackBerry PlayBook tablet
 - about, 25
 - simulator for, 107
- block elements, 228
- BlueVia platform, 539
- blur event, 479
- bookmarking (mobile browsers), 185
- bookmarklets, 453, 684
- border-image property (CSS), 409–412
- border-radius property (CSS), 287, 408–408
- brands (see specific brands)
- brand_name capability (WURFL), 317
- browsers (mobile devices)
 - audio compatibility table for, 381
 - background execution, 456–462
 - bookmarking support, 185
 - browsing types, 43
 - canvas compatibility table, 364
 - client-side detection, 284–299
 - coding with JavaScript for, 452–455
 - contenteditable attribute, 250
 - direct versus cloud-based, 47
 - document margins and, 394
 - file upload selectors and, 257, 260–263
 - fragmentation and, 50–60
 - icon display compatibility table, 186
 - image format compatibility table, 336
 - JavaScript libraries, 485–487
 - key features, 87
 - legacy, 81
 - market statistics, 60
 - MIME types supported, 710
 - mobile design strategy, 127–140
 - multipage browsing, 48
 - navigator object properties compatibility table, 298
 - platform detection, 298–299
 - preinstalled, 73–82
 - pseudo-browsers, 69
 - quantity of simultaneous touches comparison table, 569
 - reflow layout engines, 46
 - server-side detection, 300–328
 - target attribute, 231
 - testing, 177
 - useful keys for, 481–483
 - user-agent string and, 302–303, 307, 309
 - user-installable, 82–86
 - vendor prefixes and, 286–290
 - video compatibility table for, 378
 - video element and, 372
 - web engines and, 49
 - WebGL compatibility, 363
 - zoom experience, 45
- BrowserStack service, 112
- Browshot tool, 114
- Budiu, Raluca, 145
- buttons, 242

C

- C# language, 308
- cache capability group (WURFL), 316
- cache manifest file (see manifest file)
- cached event, 507
- calc() function, 448
- Calendar API, 594, 640
- camera access on mobile devices, 590–593
- Camera API, 640, 649
- Can I Use website, 282
- canvas element
 - 2D context, 357
 - about, 357
 - advanced features, 359
 - as background, 360
 - compatibility table, 364
 - drawing methods, 358
 - exporting drawings, 361
 - lines and strokes, 358

- as native code, 359
- pixel density and, 361
- WebGL and, 362
- capabilities (WURFL), 316–318
- Capture API (Cordova/PhoneGap), 593, 649
- Carakan JavaScript engine, 50
- CardFlip pattern, 436–438
- carrier network location APIs, 539
- Cascades UI layer, 26
- cascading menus, 499
- Chakra JavaScript engine, 50
- Champeon, Steven, 132
- change event, 479
- charCode property, 482
- chargingchange event, 593
- chargingtimechange event, 593
- Charles Web Debugging Proxy, 674
- charset encoding, 163
- checkboxes, 236, 241
- checking event, 507
- Chitika website, 61
- Chrome (Google)
 - Android browser and, 74
 - debugger for Android platform, 681
 - debugging tool, 673
 - HTML5 apps engine, 624
 - HTML5 web apps and, 66
 - as pseudo-browser, 69
 - Ripple tool, 104
- Chromebook (netbook), 8
- cHTML (compact HTML), 158
- ciUI library, 487
- clear property (CSS), 405
- clearRect() method (Canvas API), 358
- click event, 478
- client-side detection
 - about, 284
 - CSS fallbacks, 285
 - debugging, 678–685
 - HTML fallbacks, 284
 - JavaScript fallbacks, 290–292
 - Modernizr library, 292–297
 - platform detection, 298–299
 - polyfills, 297
 - vendor prefixes, 286–290
- client-side location techniques
 - A-GPS, 530
 - about, 530
 - cell information, 531
 - GPS, 530
 - WiFi Positioning System, 531
- client-side storage
 - about, 511
 - debugging, 521
 - FileSystem API, 520
 - IndexedDB API, 515, 518–519
 - user options regarding, 521
 - Web SQL Database API, 514–517
 - Web Storage API, 511–514
- ClientLocation object, 540
- closePath() method (Canvas API), 358
- cloud-based browsers, 47, 309–311, 454
- Cocoa Touch framework, 12
- code development tools
 - about, 89
 - Adobe Dreamweaver, 89
 - Adobe Edge tools, 90
 - Eclipse, 91
 - Microsoft Visual Studio, 91
 - Microsoft WebMatrix, 91
 - for native web apps, 91
- codecs, audio and video, 368–369, 370
- column-count property (CSS), 404
- column-gap property (CSS), 404
- column-rule property (CSS), 404
- column-width property (CSS), 404
- communication, network (see network communication)
- compact HTML (cHTML), 158
- Compass APIs, 583, 649
- Compass object (Cordova/PhoneGap)
 - clearHeading method, 583
 - getCurrentHeading method, 583
 - watchHeading method, 583
- compassneeds calibration event, 580
- compressing files, 690
- conditional expressions, 385
- conditional resource loader (Modernizr), 294
- config.xml file (WebWorks), 626, 634
- configuration standards, 601
- confirm dialog (JavaScript), 467
- Connection object, 584, 584
- constraints validation API, 274
- constructor functions, 291
- Contacts APIs, 594, 640, 649
- contacts, adding to phonebooks, 559–560
- containers, audio and video, 368–369

- content (HTML5)
 - about, 228
 - accessibility, 232
 - block elements, 228
 - frames, 229
 - hyperlinks, 230–232
 - lists, 228
 - tables, 229
- content attribute (meta tag), 461
- content delivery
 - about, 655
 - application and games delivery, 662–669
 - file delivery, 658–662
 - MIME types, 655–657
- content property (CSS), 415
- Content-disposition header, 657
- Content-Type header (HTTP), 163
- contenteditable attribute, 250
- context for mobile devices
 - defining, 130
 - detecting server-side, 307–309
 - HTTP headers and, 304
- controls attribute, 373, 379
- cookie management, 476
- coords property, 536
- Cordova WebView, 646
- CordovaJS API, 647
- CORS (Cross Origin Resource Sharing), 523
- cors property, 523
- createImageData() method (Canvas API), 358
- CreateJS toolkit extension, 361, 380
- Creative Cloud Services (Adobe), 115
- Crockford, Douglas, 600
- Cross Origin Resource Sharing (CORS), 523
- CSS
 - common patterns
 - appearance override, 421
 - backgrounds, 413
 - border image, 409–412
 - content, 415
 - cursor management, 416
 - display properties, 404–408
 - highlight color, 421
 - opacity, 416
 - pseudoclasses, 413
 - rounded corners, 408–408
 - scrollable areas, 414
 - selection management, 417–419
 - touch callout, 420
 - defining extensions, 172–176
 - Device Adaptation standard, 203
 - fallback mechanisms, 285, 288
 - Modernizr properties for, 295
 - multiple screen densities, 345
 - noncompatible tags and, 172
 - resetting files, 393
 - text formatting considerations, 395–404
 - vendor prefixes and, 286–290, 291
 - viewport declaration and, 211
 - where to insert, 383
- CSS 3 conditionals, 286
- CSS Animation module, 442–446
- css capability group (WURFL), 317
- CSS exclusions and shapes module, 447
- CSS filter effects specification, 446
- CSS Flexible Box Layout, 407
- CSS Grid Layout layout mechanism, 408
- CSS preprocessors, 288–289
- CSS regions module, 447
- CSS selectors, 392
- CSS Sprites
 - about, 422
 - alternatives to, 427
 - samples and compatibility, 422–425
- CSS transforms standard, 433–438
- CSS transitions, 439–441
- CSS3
 - about, 176
 - RWD and, 134
 - validation pseudoclasses, 273
- CSS3 Image Values specification, 428–429
- CSS3 media queries
 - about, 385–386
 - all versus only screen, 388
 - CSS conditionals and, 286
 - IE 10 snap state, 390–391
 - multiple screen densities and, 347
 - RWD and, 134
 - standard features attributes, 386
 - useful queries, 388–389
 - viewports and, 212
- CSS3 values and units module, 448
- CSS4 media queries, 390
- CSSPrefixer tool, 290
- css_spriting capability (WURFL), 318
- cubic-bezier() function, 440
- Cunningham, Katie, 233
- cursor management (CSS), 416

cursor navigation, 44, 337

cursor-event-mode meta tag, 215, 568

D

DASH (Dynamic Adaptive Streaming over HTTP), 376

data URLs, 334–335, 336

data-blackberry-end-selection-on-touch attribute, 249

data-win-control attribute, 616

datalist tag, 247

datauri attribute, 335

WebView data type, 464

date input fields, 255, 297

DatePicker control, 617

dc: URI scheme, 556

dd tag, 228

DDR (Device Description Repository), 313–328

Debug Bridge tool (Android), 682

debugging

- about, 671

- client-side detection, 678–685

- client-side storage, 521

- JavaScript on mobile browsers, 455

- markup, 676–677

- server-side detection, 671–676

defer attribute, 453

definition lists, 228

delayed linking model of file delivery, 658

deleting

- margins, 394

- packages, 505

design strategies

- context for mobile device, 130

- for navigation, 140

- progressive enhancement technique, 132–134

- Responsive Web Design, 134–138

- RESS, 139–140

- server-side adaptation, 131

- for user experience, 141–152

- when to get out of the browser, 127–129

desktop websites, 36

DetectFree (Movila), 327

detection, feature and device (see feature and device detection)

DetectRight engine, 326

Deutsche Telekom R&D Laboratories, 455

development tools

- production environment and, 124–126

- for testing, 91–124

- working with code, 89–91

Device API (Cordova/PhoneGap), 649

Device Central (Adobe), 111

Device Description Repository (DDR), 313–328

device detection (see feature and device detection)

device libraries (see specific libraries)

device pixel ratio, 55, 346, 347

device-aspect-ratio attribute, 387

device-height attribute, 387

device-pixel-ratio attribute, 347

Device-Stock-UA header (HTTP), 307

device-width attribute, 387

DeviceAnywhere (Keynote), 120–122

DeviceAtlas product, 324–326

DeviceMap (Apache), 328

devicemotion event, 580

deviceorientation event, 580, 583

DeviceOrientation Event specification

- about, 579

- moving a ball, 581–582

devicePixelRatio property, 346

deviceready event, 647

device_os capability (WURFL), 317

device_os_version capability (WURFL), 317

dialogs, JavaScript, 467–469

Digital Rights Management (DRM), 659

direct browsers, 47

direct linking model of file delivery, 658

directions

- Google Maps for Android, 542

- iOS Maps application, 544

disabled attribute, 268

dischargingtimechange event, 593

discoverability, spiders and, 696–698

display (mobile devices)

- about, 50

- full screen experience, 587

- input methods, 58–58

- physical dimensions, 53–55

- pixel aspect ratio, 57

- pixel density ratio, 55

- resolution, 51–53

display capability group (WURFL), 316

display property (CSS), 404, 422

- distribution
 - about, 601
 - of iOS web apps, 610, 664
 - of Mozilla OWA web apps, 622
 - of native web apps, 650
 - of WebWorks apps, 632
- DOCTYPE (Document Type Declaration)
 - about, 160
 - standards preferences for, 161–164
 - WML and, 156
- document body (HTML5)
 - about, 224
 - main structure, 225–226
- document object
 - getElementById() function, 463
 - hidden property, 457
 - querySelector() function, 463
 - querySelectorAll() function, 463
 - visibilityState property, 457
- Document Object Model (DOM)
 - about, 463
 - Web Workers specification and, 465
- Dolphin browser, 79
- Dolphin browser, 86
- DOM (Document Object Model)
 - about, 463
 - Web Workers specification and, 465
- DOM mutation observer, 253
- domain considerations, 125
- DOMCharacterDataModified event listener, 253
- DOMNodeInserted event listener, 253
- DOMNodeRemoved event listener, 253
- dotMobi TLD, 42
- dots per inch (DPI), 53
- double exclamation mark, 291
- double-tap action, 478
- download manifest file, 664
- downloading event, 507
- DPI (dots per inch), 53
- Dragonfly debugging tool, 682
- drawImage() method (Canvas API), 358
- Dreamweaver (Adobe), 89, 157
- DRM (Digital Rights Management), 659
- Droid devices, 16, 28
- dt tag, 228
- Dynamic Adaptive Streaming over HTTP (DASH), 376
- dynamic definition (MIME types), 657
- E**
 - e-book readers, 72
 - Eclipse Project, 91
 - ECMAScript, 451
 - ECMAScript Mobile Profile (ESMP), 451, 452
 - Edge Animate (Adobe), 90, 442
 - Edge Code (Adobe), 90
 - Edge Inspect (Adobe), 90, 115–116
 - Edge Reflow (Adobe), 90
 - 80/20 law, 140
 - Ejecta iOS implementation, 359
 - email, sending, 556, 699
 - embedding
 - objects, 378
 - SVG in XHTML, 354
 - Emoji (images), 340
 - emoticons, 342
 - emptyok attribute, 276
 - emulators
 - comparison chart, 112
 - emulators, 92
 - remote emulation services, 112
 - testing mobile apps with, 92–114
 - WML, 157
 - encoding, charset, 163
 - Enyo framework, 490
 - error event, 507
 - error handling
 - Geolocation API, 536
 - production environment and, 125
 - ESMP (ECMAScript Mobile Profile), 451, 452
 - Event API, 627
 - event handling
 - emulating double click action, 478
 - managing events, 477–483
 - preventing default behavior, 483
 - XMLHttpRequest object and, 523
 - Event Source, 525–527
 - events, 477
 - (see also specific events)
 - animation, 446
 - double-tap action, 478
 - managing, 477
 - pointer, 569–572
 - preventing default behavior, 483
 - touch, 565–569
 - exclamation mark, 291
 - exporting canvas drawings, 361

F

- Facebook Connect for Mobile Web, 706
- Facebook site, 70
- facetime: URI scheme, 559
- fallback mechanisms
 - application cache, 504
 - CSS, 285, 288
 - HTML, 284
 - JavaScript, 290–292
 - recommendations for, 281
 - video element, 374
- feature and device detection
 - about, 279
 - client-side detection, 284–299
 - informational websites, 281–284
 - possible problems and their solutions, 279–281
 - server-side detection, 300–328
- feature phones, 4
- Fiddler Connection Simulator, 676
- fidelizing users, 701–704
- fieldset tag, 236
- 51 Degrees website, 61, 328
- File API, 585–587, 649
- file compression, 690
- file delivery
 - content delivery and, 658
 - direct linking model, 658, 658
 - OMA Download standard, 659–662
- File I/O API, 627
- file selection fields, 257–263
- FileReader object
 - readAsBinary() method, 586
 - readAsDataURL() method, 586
 - readAsText() method, 586
- FileSystem API, 520
- fill() method (Canvas API), 358
- fillRect() method (Canvas API), 358
- fillStyle property, 358
- fillText() method (Canvas API), 359
- filter effects, 446
- find dialog (JavaScript), 467
- Firebug Lite plug-in, 685
- Firefox platforms
 - emulator for, 111
 - Firebug Lite plug-in, 685
 - Firefox OS, 30
 - form control attributes, 268
 - HTML5 web apps and, 66
 - remote debugging, 681
 - user agent tricks, 307
 - user-installed browsers, 86
- fixed position, 406
- Flash Player (Adobe), 123, 365–366
- flash_lite capability group (WURFL), 317
- flash_lite_version capability (WURFL), 318
- Flexbox technique, 407
- Fling, Brian, 145
- flip gesture, 572
- FlipView control, 617
- Float32Array data type, 464
- floating bars, 497–498
- floating positions, 405
- FLV container, 368
- Flyout control, 617
- fl_browser capability (WURFL), 318
- focus event, 479
- focus navigation, 43, 337
- focus pseudoclass (CSS), 413
- focus() function, 472
- @font-face declaration, 397–398
- font-family property (CSS), 395
- font-size property (CSS), 399
- form control attributes
 - about, 266
 - additional, 270
 - autocomplete, 267
 - autofocus, 267
 - disabled, 268
 - Firefox extensions, 268
 - input validation attributes, 268
 - Modernizr support, 297
 - placeholder, 266
 - readonly, 268
 - Safari extension, 268
 - XHTML MP inputmode attribute, 269
- form elements
 - about, 238
 - buttons, 242
 - checkboxes, 236, 241
 - date input fields, 255
 - file selection fields, 257–263
 - hidden fields, 243
 - non-interactive, 264–265
 - radio buttons, 241
 - range slider field, 254
 - select lists, 238–241
 - text input fields, 243–254

- form validation
 - about, 270
 - HTML5, 271–274
 - Modernizr support, 297
 - WAP CSS, 275–277
- formaction attribute, 270
- format attribute, 277
- formenctype attribute, 270
- formmethod attribute, 270
- forms (see HTML5 forms)
- formtarget attribute, 270
- FourSquare website, 533
- fragmentation
 - about, 50
 - Android platform and, 16
 - mobile browsers and, 50–60
- frames (HTML5), 229
- Full Screen API
 - about, 587
 - cancelFullScreen() method, 588
 - exitFullScreen() method, 588
 - requestFullScreen() method, 587
- full screen apps patterns, 652–654
- fullscreen pseudoclass, 588

G

- games delivery (see application and games delivery)
- Gecko rendering engine, 49
- General Packet Radio Service (GPRS), 40
- geolocation
 - about, 60, 529
 - Geolocation API, 534–541
 - IP addresses and, 309, 540–541
 - location techniques, 529–533
- Geolocation API
 - about, 534
 - defining optional attributes, 537
 - getting the position, 534
 - handling error messages, 536
 - tracking locations, 536
- Geolocation API 2.0, 538
- geolocation object
 - clearWatch() method, 537
 - getCurrentPosition() method, 534, 537, 538
 - watchPosition() method, 537, 537, 538
- gesturechange event, 574
- gestureend event, 574

- GestureEvent object
 - rotation property, 574
 - scale property, 574
- gestures
 - about, 572
 - Apple Gesture API, 574–576
 - frameworks supporting, 573
 - Microsoft Gesture API, 577–578
 - Nokia S40 web apps, 635
 - swipe, 572
- gesturestart event, 574
- GET request (HTTP), 300
- getImageData() method (Canvas API), 358
- getUserMedia() method, 590
- gif animated capability (WURFL), 318
- Global Positioning System (GPS), 60, 530
- Globalization API, 594
- Google Accounts Authentication and Authorization API, 707
- Google AdSense, 705
- Google Analytics for Mobile Websites, 704
- Google Chrome
 - Android browser and, 74
 - debugger for Android platform, 681
 - debugging tools, 673
 - HTML5 apps engine, 624
 - HTML5 web apps and, 66
 - as pseudo-browser, 69
 - Ripple tool, 104
- Google ClientLocation object, 540
- Google Code Blog, 248, 496
- Google Glass project, 8
- Google Maps API, 546–548
- Google Maps application, 541
- Google Maps Static API, 546, 549–550
- Google Mobile operating system
 - about, 13
 - UI pattern, 248
- Google Nexus devices, 16
- Google Places API, 533
- Google Play Store, 14, 98, 651
- Google search engine, 695
- Google Voice on Android, 559
- Google+ API, 707
- google.streetview: protocol, 542
- Googlebot-Mobile, 696
- gotpointercapture event, 570
- GPRS (General Packet Radio Service), 40
- GPS (Global Positioning System), 60, 530

- graceful degradation strategy, 132
- gradients, defining as image functions, 428–429
- GSM Association, 532, 539
- gyroscopes, 579

H

- H.263 codec, 368
- H.264 codec, 368
- Hammer.js library, 573
- Handbrake transcoder, 369
- Handheld Device Markup Language (HDML), 40, 153
- HAR standard, 687
- hashchange event, 471
- HDML (Handheld Device Markup Language), 40, 153
- heading structure (HTML5)
 - about, 181
 - changing navigation method, 215
 - hiding URL address bar, 218
 - Home Screen icons, 187–199
 - integrating native apps, 220–224
 - removing automatic links, 216
 - sharing meta data, 216–218
 - title, 181–182, 191
 - viewport declaration, 200
 - website icons, 183–186
- height attribute
 - CSS3 media queries, 387
 - img element, 336
 - viewports, 205, 211
- hidden fields, 243
- hiding URL address bar, 218
- highlight color, 421
- history object
 - about, 470–471
 - back() method, 470
 - forward() method, 470
 - go() method, 470
 - pushState() method, 470
 - reload() method, 506, 653
 - replaceState() method, 470
 - state property, 470
- HLS (HTTP Live Streaming), 376–377
- Home Screen icons, 187–199
- home screen widget (MiniView), 642
- hover attribute (CSS4), 390
- HP, Palm devices and, 29
- href attribute, 230

- .htaccess file extension, 656
- HTC devices, 29
- HTML
 - debugging considerations, 676–677
 - DOCTYPE for, 162
 - events supported, 477
 - fallback mechanisms, 284
 - preferred MIME type, 162
- HTML Media Capture, 259
- HTML5
 - APIs for mobile web apps, 595
 - boilerplate template for, 227
 - compatibility levels, 177–179
 - content for, 228–233
 - creating templates, 170
 - debugging considerations, 676–677
 - DOCTYPE for, 162
 - document body, 224–226
 - fallback mechanisms, 284
 - heading structure, 181–224
 - history of, 42, 169
 - JavaScript APIs, 466
 - Modernizr properties for, 295
 - new elements in, 172
 - offline web apps, 501–511
 - preferred MIME type, 162
 - script extensions, 453
 - storage limits and, 513
 - syntax rules, 171
 - validating forms, 271–274, 297
- HTML5 elements
 - canvas, 357–365
 - client-side detection and, 284
 - for forms, 238–265
 - new elements, 172
 - picture, 350
 - video, 370–376
- HTML5 forms
 - about, 235
 - designing, 235–237
 - elements in, 238–265
 - form control attributes, 266, 297
 - validating, 270–277, 297
- HTML5 Test website, 177
- HTML5 web apps
 - about, 64–66
 - e-books and, 72
 - Facebook and, 70
- HtmlControl control, 617

- html_web_4_0 capability (WURFL), 318
- html_wi_oma_xhtmlmp_1_0 capability (WURFL), 318
- HTTP (HyperText Transfer Protocol)
 - about, 300
 - delivering video, 370
 - request process, 300–307
 - SPDY and, 301
 - streaming support, 376–377
 - User Agent Profile and, 306
 - user-agent string, 302–303
- HTTP compression, 690
- HTTP Dynamic Streaming (Adobe), 376
- HTTP headers
 - common, 301
 - Opera Mini supported, 310
 - reading, 308
 - Xpress Browser supported, 311
- HTTP Live Streaming (HLS), 376–377
- HTTP sniffing, 673, 687
- http: protocol, 561
- hybrid apps, 69–72
- hyperlinks (HTML5), 230–232
- HyperText Transfer Protocol (see HTTP)

I

- i-mode XHTML, 230
- LAB (Interactive Advertising Bureau), 706
- IANA (Internet Assigned Number Authority), 554
- ICANN (Internet Corporation for Assigned Names and Numbers), 42
- IceCast Server, 377
- ICO format, 183
- IcoMoon service, 343
- icons
 - Home Screen, 187–199
 - website, 183–186
- iDEN networks, 556
- Identity API, 627
- Idle API, 594
- iframe tag, 229, 414
- IIS (Internet Information Server), 300, 527, 656
- image maps, 337
- image-rendering property (CSS), 340
- image-set() function, 285, 349
- images
 - Adobe Flash, 365–366
 - CSS Sprites alternatives, 427
 - formats supported, 331–336
 - img element, 336
 - joining, 427
 - local pictograms, 340–342
 - masked, 432
 - memory consumption and, 332
 - MIME types supported, 710
 - multiple screen densities, 342–365
 - responsive, 138, 337–340
- image_format capability group (WURFL), 316
- image_inlining capability (WURFL), 318, 335
- IMEI (International Mobile Equipment Identity), 304
- iMessage service, 558
- img element
 - multiple screen densities, 344
 - SVG and, 355
 - usage considerations, 336
- importScripts() function, 466
- :in-range pseudoclass, 273
- IndexedDB API, 515, 518–519
- indoor location
 - about, 529
 - server-side location techniques and, 533
- infinite lists, 146
- initial-scale attribute, 205
- inline images, 334–335, 427
- inline SVG, 355
- input element
 - date input fields and, 255
 - file selection fields and, 257
 - range slider field and, 254
 - text input fields and, 244
- input methods for mobile devices, 58–58
- input validation attributes, 268
- inputformat attribute, 277
- inputmode attribute, 269
- Int32Array data type, 464
- Intel (Adobe)
 - native web apps for, 91
- Intel computers, 94
- intents (Android), 562
- Interactive Advertising Bureau (IAB), 706
- International Mobile Equipment Identity (IMEI), 304
- Internet Assigned Number Authority (IANA), 554
- Internet Corporation for Assigned Names and Numbers (ICANN), 42

- Internet Explorer
 - about, 76
 - snap state, 390–391
 - Internet Explorer Mobile, 214
 - Internet Information Server (IIS), 300, 527, 656
 - :invalid pseudoclass, 273
 - Invoke API, 627
 - IonMonkey JavaScript engine, 50
 - iOS AppStore, 650
 - iOS Developer Program, 650
 - iOS platform
 - about, 10–13
 - application and games delivery, 663–664
 - creating web apps for, 602–612
 - detecting if application is installed, 561
 - detection problem for devices, 305
 - devices supported per version, 11
 - HTML5 web apps and, 65
 - Maps application, 543, 560
 - reference movies for, 369
 - Remote Web Inspector, 678–680
 - simulator for, 99–104
 - Smart App Banner service, 220–222, 561
 - URL schemes, 560–562
 - IP addresses
 - cloud-based browsers and, 310
 - geolocation and, 309, 540–541
 - ipconfig command for, 114
 - reading, 308, 540
 - server-side location techniques, 532
 - iPad device
 - about, 9
 - iOS detection problem, 305
 - iOS Simulator and, 101
 - user-agent string and, 304
 - ipconfig command, 114
 - iPhone device
 - about, 9, 42
 - iOS detection problem, 305
 - iOS Simulator and, 101
 - local pictograms, 341
 - Retina display, 55
 - streaming support, 377
 - user-agent string and, 304
 - iPhone Safari Viewport Scaling Bug, 210
 - iPod device
 - about, 10
 - iOS detection problem, 305
 - iScroll JavaScript solution, 406
 - ISO pictograms, 342
 - is_smarttv capability (WURFL), 317
 - is_tablet capability (WURFL), 317
 - is_transcoder capability (WURFL), 318
 - is_wireless_device capability (WURFL), 317
 - itms-services: protocol, 664
 - iTunes Affiliate Program, 221
 - iTunes Link Maker, 221, 663
 - iTunes software, 10
 - iUI library, 487, 492–493
 - iWebInspector tool, 680
 - iWebKit library, 487
- ## J
- j2me capability group (WURFL), 317
 - JAD (Java Application Descriptor) files, 667–668
 - JägerMonkey JavaScript engine, 50
 - Japanese mobile web, 82
 - JAR (Java ARchive) files, 667
 - Jasmine browser, 79
 - Java ME (Micro Edition)
 - about, 21
 - application and games delivery, 667–669
 - MIDlet OTA installation method, 659
 - Java Servlets (JSP), 308, 657
 - Java Virtual Machine (JVM), 13
 - JavaScript APIs
 - about, 290, 564
 - battery status, 593
 - camera access, 590–593
 - file management, 585–587
 - full screen experience, 587
 - gestures, 572–578
 - HTML5 support, 466
 - native web apps and, 466
 - network information, 583
 - sensors, 579–583
 - touch devices, 564–572
 - vibrating mobile devices, 593
 - web notifications, 588–590
 - JavaScript Debug Toolkit (JSDT), 684
 - JavaScript engines, 50
 - JavaScript language
 - about, 50, 451
 - applicationCache object, 506–508
 - background execution, 456–462
 - battery consumption, 455
 - changing titles, 476

- coding for mobile browsers, 452–455
- cookie management, 476
- debugging considerations, 455
 - as dynamic application engine, 653
- event handling, 477–483
- fallback mechanisms, 290–292
- focus and scroll management, 472
- history and URL management, 470–471
- manipulating windows, 471
- mobile UI patterns, 495–500
- navigator object, 298–299
- performance optimization, 692
- profiling considerations, 455
- progressive enhancement and, 134
- prompt dialog, 254
- standard behavior, 467–483
- standard dialogs, 467–469
- supported technologies, 463–466
- timer support, 473–476
- UI controls, 487–500, 616
 - vendor prefixes and, 291
- JavaScript libraries, 484–487
- JavaScript Object Notation (JSON), 464
- javascript: URL protocol, 453
- Jehl, Scott, 210, 219
- jGestures framework, 573
- jHere framework, 550
- JIT (Just in Time) compiler, 50
- Jo framework, 573
- Jobs, Steve, 396
- join images, 427
- Jolley, Charles, 692
- joules (energy consumption), 455
- JPS (JPEG Stereoscopic) format, 333
- jQMobi library, 486
- jQTouch library, 487, 493
- jQuery Mobile, 255, 488, 489
- jqUI library, 487
- JSConsole tool, 683
- JSDT (JavaScript Debug Toolkit), 684
- JSON (JavaScript Object Notation), 464
- JSON object
 - parse() function, 464
 - stringify() function, 464
- JSONP (JSON with Padding), 340, 524
- JSP (Java Servlets), 308, 657
- Just in Time (JIT) compiler, 50
- JVM (Java Virtual Machine), 13

K

- keyCode property, 481
- keydown event, 481
- @keyframes at-rule (CSS), 443–446
- Keynote DeviceAnywhere, 120–122
- Keynote MITE tool, 112, 672
- keypress event, 481
- keyup event, 481, 483
- Khronos Group, 362
- KHTML engine, 49
- Kindle Fire tablet, 28, 311
- Kosmaczewski, Adrián, 489

L

- label tag, 236
- Landmarks API, 640
- Lawnchair store, 517
- Lawson, Bruce, 287
- layout engines
 - reflowing pages, 46
 - WebKit, 49
- layout techniques
 - CSS Grid Layout, 408
 - Flexbox, 407
 - multicolumn layout, 403
- LBS (location-based services)
 - about, 529
 - Geolocation API, 534–541
 - location techniques, 529–533
- LBS Sprint API, 539
- legacy mobile browsers, 81
- legend tag, 236
- LESS client-side processor, 289
- levelchange event, 593
- LG Mobile devices
 - about, 28
 - preinstalled browser, 79
- libraries (see specific libraries)
- linear-gradient() function, 428
- lineTo() method (Canvas API), 358
- lineWidth property, 358
- link pseudoclass (CSS), 413
- list attribute, 248
- lists (HTML5)
 - about, 228
 - navigation lists, 231
 - select lists, 238–241
- Live Reload, 654

- load event, 477
- LocAid services, 539
- localapp: scheme, 563
- localStorage object, 511–514
- Location API, 640
- location object
 - about, 470–471
 - hash property, 470
 - href property, 470, 471, 553
- Location Orange API, 539
- location techniques
 - about, 529
 - accuracy in, 529
 - asking the user, 533
 - client-side techniques, 530–531
 - indoor location, 529
 - server-side techniques, 532–533
- location-based services (LBS)
 - about, 529
 - Geolocation API, 534–541
 - location techniques, 529–533
- Logging API, 640
- loop attribute, 374
- lostpointercapture event, 570
- Lungo framework, 488, 573

M

- m subdomain, 41
- Maemo platform
 - about, 24
 - preinstalled browser, 81
- magnetometers, 579
- magometers, 60
- mailto: protocol, 556
- manifest file
 - about, 501, 502, 620
 - CACHE: section, 503
 - FALLBACK: section, 504
 - NETWORK: section, 503
 - reusing, 503
- maps
 - native map applications, 541–545
 - showing, 546–551
- Maps application (iOS), 543, 560
- maps: protocol, 543, 561
- Marcotte, Ethan, 134
- margins, deleting, 394
- market share information, 60
- marketing_name capability (WURFL), 317

- markup capability group (WURFL), 316
- markup debugging, 676–677
- markup languages
 - delivering markup, 160–164
 - historical, 153
- marquee element, 174
- masked images, 432
- Massive Operator Identification Platform, 309
- matrix code, 699
- matrix() CSS function, 433
- matrix3d() CSS function, 433
- max attribute, 254, 268, 271
- max-height attribute, 211
- max-width attribute, 211
- max-zoom attribute, 212
- maximum-scale attribute, 205
- maxlength attribute, 248, 270
- max_image_height capability (WURFL), 318
- max_image_width capability (WURFL), 318
- measuring performance optimization, 686–688
- media (see audio; images; video)
- Media API (Cordova/PhoneGap), 649
- media attribute, 350, 384
- Media Capture and Streams API, 590
- Media Management API, 640
- media queries (see CSS3 media queries)
- MeeGo platform
 - about, 24
 - apple-touch icon and, 191
 - emulator for, 111
 - preinstalled browser, 78
- memory consumption, image size and, 331
- Menu control, 617
- menu object (Symbian webapps)
 - hideSoftkeys() method, 641
 - setLeftSoftkeyLabel() method, 641
 - setRightSoftkeyLabel() method, 641
 - showSoftkeys() method, 641
- menubutton event (Cordova/PhoneGap), 648
- MenuItem object, 641
- menus, cascading, 499
- message event, 465
- Messaging API, 594, 627, 640
- meta data, sharing, 216–218
- meter element, 264
- MIB (Mobile Internet Browser), 81
- Microsoft Gesture API, 577–578
- Microsoft Pointer Events specification, 569–572
- Microsoft Smooth Streaming, 376

- Microsoft Surface tablets, 19
- Microsoft Visual Studio, 91
- Microsoft WebMatrix, 91
- Microsoft Windows Phone Store, 651
- Microsoft Windows Store, 651
- MIDlet-Delete-Notify attribute, 668
- MIDlet-Install-Notify attribute, 668
- MIDlet-Jar-URL attribute, 668
- MIME types
 - about, 160
 - audio and video, 711
 - content delivery and, 655
 - CSS preferences for, 160
 - dynamic definition, 657
 - image, 710
 - markup and script, 709
 - mobile content, 710
 - standards preferences for, 161–164
 - static definition, 655–656
 - web apps, 711
- min attribute, 254, 268, 271
- min-device-pixel-ratio attribute, 347
- min-height attribute, 211
- min-width attribute, 211
- min-zoom attribute, 212
- minimum-scale attribute, 205
- MiniView (home screen widget), 642
- mirror effects (reflection), 431
- MITE (Mobile Interactive Testing Environment), 112, 672
- mixins, 289
- MMS (Multimedia Messaging Service), 558, 659
- mms capability group (WURFL), 317
- mms: URI scheme, 558
- .mobi extension, 42
- mobile devices, 279
 - (see also feature and device detection)
 - about, 38
 - background execution, 456–462
 - battery consumption, 455
 - brands, models, and platforms, 9–32
 - browsing experience, 43–50
 - categories of, 3–9
 - client-side detection, 284–299
 - coding with JavaScript for, 452–455
 - commonly implemented width values, 207
 - delivering markup to, 160–164
 - device libraries, 313–328
 - features of, 1–3
 - input methods for, 58–58
 - JavaScript libraries, 485–487
 - key features, 60
 - market statistics, 60
 - MIME types supported, 710
 - non-phone, 7
 - quantity of simultaneous touches comparison table, 569
 - server-side detection, 300–328
 - technical information, 32
 - usage statistics on, 37
 - useful keys for, 481–483
- Mobile First concept, 140
- Mobile Interactive Testing Environment (MITE), 112, 672
- Mobile Internet Browser (MIB), 81
- Mobile Marketing Association, 705
- mobile markup checker, 677
- mobile phones, 3
- mobile SEO
 - about, 695
 - how users find you, 698–701
 - spiders and discoverability, 696–698
 - user fidelizing, 701–704
- Mobile Sitemaps, 696
- Mobile Tiny URL, 163, 699
- mobile web, 89
 - (see also development tools)
 - advertisements on, 705
 - APIs for, 595
 - differences in, 38
 - gathering statistics for, 704–705
 - history of, 39–43
 - Japanese, 82
 - myths of, 35–38
 - native apps and, 37
 - navigation design tips, 140
 - politics of, 159–160
 - social features on, 706–707
 - testing considerations, 122
 - user experience in, 141–152
- Mobile Web Best Practices specification, 677
- Mobile Web Best Practices website, 145
- Mobile Web Initiative, 177
- Mobile Web Library (MWL), 635
- MobileHTML5.org website, 282
- MobilePerf bookmarklet, 454
- mobile_browser capability (WURFL), 317

- mobile_browser_version capability (WURFL), 317
- Mobilytics website, 705
- modal pop-up windows, 416
- models (see specific models)
- model_name capability (WURFL), 317
- Modernizr library
 - about, 292–294
 - conditional loading, 294
 - linking to video files, 370
 - local pictograms, 341
 - polyfills and, 297
 - properties supported, 294–297, 518
- monetizing your website, 705
- Montage framework, 491
- Moore, Andy, 311
- Motorola devices, 27
- Motorola Mobile Internet Browser (MIB), 81
- mousedown event, 573
- mousemove event, 573
- mouseout event, 479
- mouseover event, 479
- mouseup event, 573
- mousewheel event, 480
- MOV container, 368
- moveTo() method (Canvas API), 358
- Movila DetectFree, 327
- Movila Detection, 327
- mozactionhit attribute, 268
- Mozilla Open Web Apps, 619–623
- Mozilla WebAPI initiative, 594
- MP3 audio format, 379
- MPEG-4 containers, 368
- msApplication-Arguments meta tag, 223
- msApplication-badge meta tag, 198
- msApplication-ID meta tag, 223
- msApplication-MinVersion meta tag, 224
- msApplication-OptOut meta tag, 224
- msApplication-PackageFamilyName meta tag, 223
- msApplication-TileColor meta tag, 196
- msApplication-TileImage meta tag, 196
- mscontentzoom event, 578
- MSGesture object, 577
- MSGesture() function, 577
- msgesturechange event, 578
- msgestureend event, 578
- msgesturehold event, 577
- msgesturestart event, 578

- msgesturetap event, 578
- msinertiastarted event, 578
- multicolumn layout, 403
- multiline text control, 248–248
- Multimedia Message Composer, 558
- Multimedia Messaging Service (MMS), 558, 659
- multipage browsing, 48
- multiple screen densities
 - about, 342
 - background images and, 345
 - canvas element and, 357–365
 - providing image alternatives, 346
 - providing single image, 344
 - SVG and, 350–357
 - vector-based solutions, 343
- multitouch navigation, 45
- muted attribute, 374
- MWL (Mobile Web Library), 635
- Myriad browser, 81

N

- \n (newline) escape special character, 469
- native apps
 - design strategy, 127–140
 - HTML5 web apps and, 64
 - integrating, 220–224
 - mobile web and, 37
 - platform-specific solutions, 71
 - URI association from, 563
 - user experience considerations, 141–152
 - web shortcuts and, 702
 - Windows 8 support, 613–619
- native map applications
 - about, 541–545
 - showing maps, 546–551
- native web apps
 - about, 69–72
 - code development tools for, 91
 - creating, 644–650
 - design strategy, 127–140
 - distributing, 650
 - HTML5, 64–66
 - JavaScript APIs, 466
 - MIME types supported, 711
 - user agent tricks, 307
 - user experience considerations, 141–152
- navigation (mobile browsers)
 - bad example of, 130
 - changing default method for, 215

- Google Maps for Android, 542
 - image maps and, 337
 - iOS Maps application, 544
 - multiple views and, 652
 - navigator object, 298–299
 - types of, 43
 - usability tips, 140
 - Windows 8 web apps, 617
 - navigation link menus, 226
 - navigation lists, 231
 - Navigation Timing API, 688
 - navigator object
 - about, 298–299
 - appName property, 298
 - appVersion property, 298
 - geolocation property, 534
 - maxTouchPoints property, 570
 - onLine property, 507
 - platform property, 298
 - plugin property, 298
 - pointerEnabled property, 570
 - properties compatibility table, 298
 - userAgent property, 298
 - userLanguage property, 298
 - netbooks, 8
 - NetFront browser, 79
 - NetLimiter tool, 676
 - NetMarketShare website, 61
 - network communication
 - about, 522
 - Ajax, 522–525
 - Server Sent Events, 525–527
 - Web Sockets API, 527
 - Web Sockets protocol, 527
 - Network Information API
 - about, 583, 649
 - new specification, 584
 - old specification, 584
 - Network Link Conditioner, 674
 - network requests, reducing, 689
 - newline (\n) escape special character, 469
 - Nexus devices, 16
 - NFC tags, 701
 - Nielsen, Jakob, 145, 246
 - Node.js library, 491, 528
 - Nokia devices
 - about, 20
 - apple-touch icon and, 191
 - brands, models, and platforms, 20–24
 - preinstalled browser, 77
 - reading HTTP headers, 308
 - remote lab service, 117–119
 - web app support, 632–643
 - Xpress Web App Builder, 702
 - Nokia Here services, 550
 - Nokia Mobile Web Library, 454
 - Nokia Mobile Web Templates, 394
 - Nokia Store, 20, 637
 - Nokia Web Tools, 91, 636, 643
 - non-interactive form elements, 264–265
 - non-phone mobile devices, 7
 - noscript tag, 454
 - NOSQL database, 518
 - not not operator, 291
 - NOT operator, 386
 - Notification API (Cordova/PhoneGap), 649, 649
 - notifications, push, 462, 649
 - noupdate event, 507
 - NTT DoCoMo services, 230, 395
- ## O
- Obigo mobile browser, 81
 - object embedding, 378
 - object tag, 342
 - object_download capability group (WURFL), 316
 - obsolete event, 507
 - ODP (On Device Portals), 43
 - offline web apps
 - about, 501
 - accessing online resources, 503
 - compatibility and limits, 508–511
 - deleting the package, 505
 - JavaScript and, 506–508
 - manifest files, 502
 - updating the package, 504
 - Ogg container, 369
 - OMA (Open Mobile Alliance)
 - ESMA standard, 451
 - local pictograms, 342
 - UAProf standard, 306
 - WAP standard, 39
 - WML standard, 154
 - OMA Download standard
 - about, 659
 - download descriptor, 660
 - post-download status report, 661

- On Device Portals (ODP), 43
- onchange event, 248
- onclick event, 472, 479
- ondblclick event, 478
- one-view widget, 653
- OneAPI initiative, 532, 539
- onhide event, 643
- onkeyup event, 248
- onload event, 461, 472
- ONLY operator, 386
- only screen type, 388
- onmousedown event, 479
- onmouseover event, 479
- onmouseup event, 479
- onresize event, 480, 643
- onSelect event, 641
- onshow event, 641, 643
- opacity property (CSS), 416
- Open Handset Alliance, 13
- Open Mobile Alliance (see OMA)
- Open Search description protocol file, 702
- Open Web Apps (OWA), 619–623
- Open Web Devices, 30
- Open webOS, 30
- OpenCellID project, 531
- OpenDDR project, 328
- OpenGL ES 2.0, 362
- Openwave browser, 81
- Opera Mini browser
 - about, 37, 84
 - simulator for, 110
 - user-agent string, 309
- Opera Mobile browser
 - about, 84
 - Dragonfly debugging tool, 682
 - user agent tricks, 307
- Opera Mobile Labs, 84
- Operator Identification Platform (Massive), 309
- OPhone operating system, 31
- option element, 247
- option groups (select lists), 241
- :optional pseudoclass, 273
- OR operator, 386
- ordered lists, 228
- orientation attribute, 212, 387
- orientationchange event, 480
- OTA (Over-The-Air)
 - about, 60, 124
 - file delivery and, 658
- :out-of-range pseudoclass, 273
- output element, 264
- Over-The-Air (OTA)
 - about, 60, 124
 - file delivery and, 658
- overflow property (CSS), 414
- OWA (Open Web Apps), 619–623

P

- packages
 - about, 601
 - deleting, 505
 - manifest file and, 501
 - updating, 504
- package_name attribute, 646
- packaging standards, 601
- Page Visibility API, 457
- PageControl control, 617
- pagehide event, 458
- pageshow event, 458
- Palm devices, 29
- Passani, Luca, 313
- Passbook (Apple), 703
- password text input, 246
- pattern attribute, 268, 271
- patterns
 - for tablets, 148
 - for touch devices, 145
 - JavaScript mobile UI, 495–500
- pause event (Cordova/PhoneGap), 648
- Payment API, 594, 627
- PCapPerf tool set, 687
- pdf capability group (WURFL), 317
- Pearce, James, 583
- Perfecto Mobile, 122–124
- performance optimization
 - about, 686
 - best practices, 689–693
 - measuring, 686–688
- PERMISSION_DENIED error constant, 536
- perspective() CSS function, 433, 435
- Phantom Limb utility, 572
- Phone API, 627
- phonebooks, adding contacts to, 559–560
- PhoneGap (Adobe), 72
 - (see also Apache Cordova framework)
 - about, 72
 - native web apps for, 644–650
- PhoneGap Build service, 72, 645

- PHP language
 - base64 conversion, 334
 - detecting mobile user agents and browsers, 311
 - reading headers, 308
 - sleep() function, 527
 - WURFL Standalone API and, 319–322
- Pick Contact Intent API, 594
- pictograms
 - about, 340
 - ISO, 342
 - OMA, 342
- picture element, 350
- Pierce, Greg, 562
- PIM API, 627
- pin: URL scheme, 558
- pixel aspect ratio, 55, 57
- pixel density ratio, 55, 361
- pixels per inch (PPI), 53
- placeholder attribute, 266, 297
- platforms (see web platforms)
- playback capability group (WURFL), 317
- PlayBook tablet (BlackBerry)
 - about, 25
 - simulator for, 107
- .plist file extension, 602
- plug-ins, 650
- PNG format, 184, 639
- pointer attribute (CSS4), 390
- Pointer Events specification
 - about, 569
 - detecting pointers, 570
 - simulating touch on desktops, 572
 - touch-action style, 571
 - working with touches and clicks, 571
- Pointer Lock API, 594
- pointercancel event, 570
- pointerdown event, 570
- PointerEvent object
 - about, 571
 - currentPoint property, 571
 - hwTimestamp property, 571
 - intermediatePoints property, 571
 - isPrimary property, 571
 - pointerId property, 571
 - pointerType property, 572
 - pressure property, 572
 - tiltX property, 572
 - tiltY property, 572
 - timeStamp property, 572
- pointermove event, 570
- pointerout event, 570
- pointerover event, 570
- pointerup event, 570
- pointing_method capability (WURFL), 317
- polyfills
 - about, 281, 297
 - APNG-canvas, 333
 - local pictograms, 341
 - usage suggestions, 280
- pop-up windows, 416, 471
- popup control, 254
- portability, 2, 51
- position property (CSS), 405–406
- PositionError class, 536
- POSITION_UNAVAILABLE error constant, 536
- POST request (HTTP), 300
- postMessage() function, 465
- Power Management API, 594
- PPI (pixels per inch), 53
- pre-installed browsers, 73–82
- preferred_markup capability (WURFL), 318
- prefix-free framework, 290
- prefixes, vendor (see vendor prefixes)
- PrefixMyCSS tool, 290
- Prefixr tool, 290
- preload attribute, 374
- preprocessors, CSS, 288–289
- pressure sensors, 579
- Presto rendering engine, 49
- print dialog (JavaScript), 467
- production environment
 - about, 124
 - domain alternatives, 125
 - error management, 125
 - usage statistics, 126
 - web hosting, 125
- product_info capability group (WURFL), 316
- profiling JavaScript on mobile browsers, 455
- progress element, 264, 285
- progress event, 507
- progressive enhancement technique, 132–134
- prompt dialog (JavaScript), 467
- proxied browsers, 47, 300, 532
- Proximity Events API, 594
- pseudo-browsers, 68, 69
- pseudoclasses, 413

- Push API, 627
- push notifications, 462, 649
- Pusher service, 528
- putImageData() method (Canvas API), 358

Q

- QR Code, 114, 699
- Qt framework, 26
- quadraticCurveTo() method (Canvas API), 358
- QuoJS library, 486
- QWERTY keyboard, 58, 481

R

- radial-gradient function, 428
- radio buttons, 241
- range slider field, 254, 297
- Raphaël framework, 356
- Rating control, 617
- Rauch, Guillermo, 528
- RDA (Remote Device Access), 117
- :read-only pseudoclass, 273
- :read-write pseudoclass, 273
- Readability bookmarklet, 453
- readonly attribute, 268
- ready.mobi validator, 677
- readystatechange event listener, 523
- Real Time Streaming Protocol (RTSP), 376
- Reality display (Xperia), 55
- reducing network requests, 689
- reference movies for iOS, 369
- reflection (mirror effects), 431
- reflow layout engines, 46
- regressive enhancement technique, 133
- reloading web apps, 653
- remote debugging, 678, 681
- Remote Device Access (RDA), 117
- remote emulation services, 112
- remote labs
 - about, 116
 - Keynote DeviceAnywhere, 120–122
 - Nokia device access, 117–119
 - Perfecto Mobile, 122–124
 - Samsung device access, 119
- remote profiling, 687
- Remote Test Lab (RTL), 119
- Remote Web Inspector, 678–680, 682, 683
- rendering engines, 49
- repeating-linear-gradient() function, 428

- repeating-radial-gradient() function, 428
- Request desktop site feature, 306
- requests (HTTP), 300–307
- required attribute, 268, 271, 297
- :required pseudoclass, 273
- resetting CSS files, 393
- resize event, 480
- resolution (screen display), 51–53
- resolution attribute, 387
- resolution_height capability (WURFL), 318
- resolution_width capability (WURFL), 318
- responseType attribute, 523
- responsive images, 138, 337
- Responsive Web Design (RWD), 134–138
- RESS (Responsive Web Design + Server Side Components), 139–140
- results attribute, 245
- resume event (Cordova/PhoneGap), 648
- Retina display (iPhone), 55
- rgba() color function, 416, 421
- rich text controls, 249–253
- RIM-COD-Size attribute, 669
- RIM-COD-URL attribute, 669
- Ringmark browser testing suite, 178
- Ripple tool, 104, 104, 631
- Rohrl, Cathy, 155
- rotate() CSS function, 433, 575
- rotate3d() CSS function, 433
- rotateX() CSS function, 433
- rotateY() CSS function, 433
- rotateZ() CSS function, 433
- rounded corners, 408–408
- rss capability group (WURFL), 317
- RSS files, 702
- RTL (Remote Test Lab), 119
- RTSP (Real Time Streaming Protocol), 376
- RV container, 368
- RWD (Responsive Web Design), 134–138

S

- Safari browser
 - about, 73
 - form control attributes, 268
- SailFish OS, 24
- Samsung Apps, 14
- Samsung devices
 - about, 26
 - HTML5 web apps and, 66
 - preinstalled browsers, 79

- remote lab service, 119
- web apps support, 624
- SASS (Syntactically Awesome Stylesheets), 288
- scale() CSS function, 362, 433, 575
- scale3d() CSS function, 433
- scaleX() CSS function, 433
- scaleY() CSS function, 433
- scaleZ() CSS function, 433
- ScientiaMobile, 313
- screen display (mobile devices)
 - about, 50
 - input methods, 58–58
 - physical dimensions, 53–55
 - pixel aspect ratio, 57
 - pixel density ratio, 55
 - resolution, 51–53
- screen resolution, 51–53
- script attribute (CSS4), 390
- script element (HTML5)
 - attributes supported, 453
 - code execution and, 453
 - JSONP and, 524
- scroll event, 480, 497
- search engine optimization (see SEO)
- searchbutton event (Cordova/PhoneGap), 648
- security capability group (WURFL), 316
- select tag, 238–241
- selection management (CSS), 417–419
- selectors (CSS), 392
- Selectors API, 463
- SemanticZoom control, 617
- Sencha Touch Animator tool, 442
- Sencha Touch framework, 72, 488–489
- Sencha.io service, 339–340
- sensors
 - about, 579
 - Apache Cordova API, 583
 - compass information, 583
 - DeviceOrientation Event specification, 579–583
- Sensors API, 640
- SEO (search engine optimization)
 - about, 695
 - how users find you, 698–701
 - spiders and discoverability, 696–698
 - user fidelizing, 701–704
- Series 40 (Nokia)
 - about, 21
 - HTML5 web apps and, 66
 - preinstalled browser, 77
 - web app support, 632–637
 - WML emulators and, 157
 - Xpress Web App Builder, 702
- Server Sent Events (SSE), 525–527
- server-side adaptation mechanism, 131
- server-side detection
 - about, 300
 - cloud-based browsers, 309–311
 - debugging, 671–676
 - detecting the context, 307–309
 - device libraries, 313–328
 - HTTP, 300–307
 - mobile detection, 311
 - transcoders, 312
- server-side location techniques
 - about, 532
 - carrier connection, 532
 - indoor location, 533
 - IP addresses, 532
 - language considerations, 532
- Session and Identification API, 595
- sessionStorage object, 511–514
- SettingsFlyout control, 617
- Sharp, Remy, 683
- Short Message Service (SMS), 41, 557, 698
- simulators
 - about, 93
 - bandwidth, 674
 - comparison chart, 112
 - testing mobile apps with, 92–114
- Sitemap protocol, 696
- sizes attribute, 188
- skewX() CSS function, 434
- skewY() CSS function, 434
- SkyFire pseudo-browser, 86
- Skyhook Wireless provider, 531
- Skype application, 559
- sleep() function (PHP), 527
- Slowly App tool, 676
- Smart App Banner service, 220–222, 561
- smart TV platforms, 31
- smart zoom, 45
- Smart-Fit feature (NetFront), 79
- smartphones
 - about, 5
 - autocomplete feature, 500
 - BlackBerry simulators, 105
 - JavaScript libraries and, 484

- Request desktop site feature, 306
- screen resolutions supported, 51
- server-side adaptation mechanism, 131
- streaming support, 376
- SVG support, 351
- title compatibility table, 182
- video element and, 371
- web views and, 67
- smarttv capability group (WURFL), 317
- SMIL (Synchronized Multimedia Integration Language), 659
- Smooth Streaming (Microsoft), 376
- SMS (Short Message Service), 41, 557, 698
- sms capability group (WURFL), 317
- sms:// URI scheme, 557
- smsto:// URI scheme, 557
- snap state (Internet Explorer), 390–391
- sniffing, HTTP, 673, 687
- Social API, 595
- social features on mobile websites
 - about, 706
 - authentication and sharing APIs, 706
 - sharing content, 707
- social phones, 5
- Socket.IO framework, 528
- Sony Mobile
 - about, 27
 - preinstalled browsers, 81
- Souders, Steve, 454, 686, 687
- SoundJS API, 380
- sound_format capability group (WURFL), 317
- SPDY protocol, 301
- spiders and discoverability, 696–698
- spoofing, user agent, 671
- src attribute
 - img element, 336, 347
 - picture element, 350
 - video element, 370
- srcset attribute, 350
- SSE (Server Sent Events), 525–527
- Standard Vector Graphics (see SVG format)
- standards, 158
 - (see also specific standards)
 - current, 158
 - delivering markup, 160–164
 - DOCTYPE preference, 161–164
 - managing multiple, 159
 - MIME type preferences, 161–164
 - politics of mobile web, 159–160
 - streaming, 376
 - for web apps, 601–602
- StatCounter website, 61, 282
- static definition (MIME types), 655–656
- statistics
 - gathering for mobile web, 704–705
 - market, 60
 - mobile device purchases, 37, 61
- status detection
 - about, 457
 - Page Visibility API, 457
 - with timers, 459
- step attribute, 268
- step-end() function, 440
- step-start() function, 440
- steps() function, 440
- storage capability group (WURFL), 316
- storage event, 512
- storage, client-side (see client-side storage)
- streaming audio and video, 375, 376–377, 591
- streaming capability group (WURFL), 316
- Street View service (Google Maps), 542
- stroke() method (Canvas API), 358
- strokeRect() method (Canvas API), 358
- strokeStyle property, 358
- strokeText() method (Canvas API), 359
- submit event, 479
- svg element, 355
- SVG format
 - about, 350
 - additional information, 353
 - custom fonts versus, 344
 - embedding in XHTML, 354
 - inline, 355
 - multiple screen densities, 343
 - Raphaël framework and, 356
 - tools for, 352–353
 - web platform compatibility table, 336
 - website icons and, 184
- SVGT devices, 351
- svg_t_1_1 capability (WURFL), 318
- svg_t_1_1_plus capability (WURFL), 318
- swipe gesture, 572
- Swiper utility, 573
- Symbian Foundation, 23
- Symbian platform
 - about, 22–24
 - apple-touch icon and, 191
 - HTML5 web apps and, 66

- local applications, 563
 - preinstalled browser, 77
 - web app support, 612, 637–643
 - Synchronized Multimedia Integration Language (SMIL), 659
 - Syntactically Awesome Stylesheets (SASS), 288
 - System API, 627
 - System Information API, 640
- ## T
- tabindex attribute, 230
 - tables (HTML5), 229
 - tablets
 - about, 7
 - Amazon manufactured, 28
 - Blackberry devices, 25
 - Nokia devices, 21
 - patterns for, 148
 - Request desktop site feature, 306
 - screen resolutions supported, 51
 - SVG support, 351
 - video element and, 371
 - web views and, 67
 - Tagstand Writer app, 701
 - target attribute, 230, 231
 - target-densitydpi attribute, 205
 - TCP Socket API, 595
 - Telefónica (company), 30
 - Telerik Fiddler tool, 674
 - terminate() method, 466
 - Test Center Developer tool, 120
 - testing mobile apps
 - about, 91
 - for accessibility, 233
 - automating for devices, 122
 - emulators and simulators, 92–114
 - HTML5 considerations, 177
 - importance of, 145
 - real device testing, 114–116
 - remote labs, 116–124
 - text boxes, clearing, 495–496
 - text fill property, 403
 - text format
 - about, 395
 - font face, 397–398
 - font family, 395
 - font size, 399
 - multicolumn layout, 403
 - text adjustment for small screens, 402
 - text overflow, 400–400
 - text shadows, 399
 - text stroke and fill, 403
 - text input fields
 - about, 243–245
 - datalist tag, 247
 - multiline text control, 248–248
 - passwords as, 246
 - popup control, 254
 - rich text controls, 249–253
 - text-overflow property (CSS3), 400–400
 - text-shadow property (CSS3), 399
 - text-size-adjust property (CSS), 402
 - text-stroke property (CSS), 403
 - textarea
 - autogrowing, 496
 - multiline text control, 248–248
 - 3rd Generation Partnership Project (3GPP), 351
 - Thomas, Neil, 459
 - 3D image formats, 333
 - 3D screens, 53
 - 3GP container, 369
 - 3GP2 container, 369
 - 3GPP (3rd Generation Partnership Project), 351
 - TIMEOUT error constant, 536
 - TimePicker control, 617
 - timers
 - for animations, 475
 - JavaScript support, 473–474
 - waking up detection using, 459
 - Titanium Studio, 91
 - title (HTML5 heading), 191
 - titles (web pages)
 - about, 181–182
 - changing, 476
 - Tizen platform
 - about, 26
 - code development tools for, 91
 - emulator for, 111
 - TLD (top-level domain), 42
 - toDataURL() method, 361
 - ToggleSwitch control, 617
 - Tooltip control, 617
 - top-level domain (TLD), 42
 - touch devices
 - clear text box button, 495–496
 - CSS touch callout, 420
 - design patterns for, 145
 - JavaScript APIs and, 564

- label tag and, 236
- pointer events, 569–572
- touch events, 565–569
- user agent tricks, 307
- Touch Event API (W3C), 568
- touch events, 565
 - (see also pointer events)
 - about, 565–567
 - BlackBerry touch behavior, 568
 - disable scrolling and, 568
 - W3C, 568
- Touch Events API (Apple), 565–568
- touch navigation, 45, 337
- Touch object
 - clientX property, 567
 - clientY property, 567
 - identifier property, 567
 - pageX property, 566, 567
 - pageY property, 566, 567
 - screenX property, 567
 - screenY property, 567
 - target property, 567
- touch-action style (CSS), 571
- touch-event-mode meta tag, 215, 568
- touchcancel event, 565
- Touché library, 572
- touchend event, 565, 573
- touchenter event, 568
- TouchEvent object
 - about, 566
 - changedTouches attribute, 566
 - event types supported, 565
 - targetTouches attribute, 566
 - touches attribute, 566
- touchleave event, 568
- touchmove event, 565, 573
- touchstart event, 565, 573
- track element, 372
- transact() method, 515
- transcoders, 225, 312, 369
- transcoder_ua_header capability (WURFL), 318
- transcoding capability group (WURFL), 317
- transform functions (CSS), 433–438
- transform property (CSS), 433, 439, 575
- transform-origin property (CSS), 434
- transform-style property (CSS), 435
- transition-delay property (CSS), 440
- transition-duration property (CSS), 440
- transition-property property (CSS), 440
- transition-timing-function property (CSS), 440, 444
- transitions
 - about, 439–440
 - animation properties, 440–441
 - considerations ending, 441
- translate() CSS function, 434
- translate3d() CSS function, 434
- translateX() CSS function, 434
- translateY() CSS function, 434, 498
- translateZ() CSS function, 434
- Trident rendering engine, 49
- TunnelTilt game, 363
- TV platforms, 31
- Twitter for Websites, 707
- 2D context, 357
- type attribute
 - date input fields and, 255
 - file selection fields and, 257
 - range slider field and, 254
 - text input fields and, 244
 - video element, 371
- typed arrays, 464

U

- UAProf (User Agent Profile) standard, 306
- Ubuntu for phones operating system, 30
- UC Browser, 78
- UI pattern, 248
- Uint8Array data type, 464
- UIWebView class, 67
- Universal iPhone UI Kit library, 488
- unload event, 477
- unordered lists, 228
- updateready event, 507
- updating packages, 504
- upgradeneeded event, 519
- URI schemes
 - about, 553
 - adding contacts to phonebooks, 559–560
 - integrating with other applications, 560
 - making calls, 554
 - opening communication apps with, 558
 - sending email, 556
 - sending SMSs, 557
- URL address bar, hiding, 218
- URL schemes, 560–562
- url() function, 428
- User Agent Profile (UAProf) standard, 306

- user agent spoofing tools, 671
- User Agent Switcher plug-in, 672
- user experience
 - design considerations, 141
 - JavaScript UI libraries, 487–500
 - official UI guidelines, 149
 - tablet patterns, 148
 - touch design patterns, 145
 - what not to do, 149–152
- User Interface API, 627
- User-Agent request header (HTTP), 301–303, 307, 696
- user-agent strings
 - about, 302–303
 - cloud-based browsers, 309
 - user agent tricks, 307
- user-installable browsers, 82–86
- user-scalable attribute, 205
- user-zoom attribute, 212
- Utility API, 627
- ux_full_desktop capability (WURFL), 317

V

- V8 JavaScript engine, 50
- :valid pseudoclass, 273
- validating forms
 - about, 270
 - HTML5, 271–274
 - input validation attributes, 268
 - Modernizr support, 297
 - WAP CSS, 275–277
- validating HTML5 code, 677
- Vanilla JS framework, 485
- VBScript language, 452
- vCard file format, 559
- VDLs (Virtual Developer Labs), 121
- velocityAngular property, 578
- velocityX property, 578
- velocityY property, 578
- vendor prefixes
 - CSS and, 286–290, 291
 - JavaScript support, 291
 - @keyframes at-rule and, 443
 - prefix-free framework, 290
 - static tools for, 290
- Verizon Network API, 539
- versions
 - Android platform, 14
 - iOS platform, 11

- Vibration API, 593
- video
 - about, 368
 - camera access on mobile devices, 590–593
 - compatibility table for web platforms, 378
 - containers and codecs, 368–369
 - delivering, 370
 - MIME types supported, 711
 - new HTML5 element, 370–376
 - object embedding, 378
 - streaming, 375, 376–377
- video element
 - about, 370–373
 - advanced declarations, 374
 - audio player and, 380
 - providing fallbacks, 374
 - streaming example, 377, 591
- video player API, 375
- ViewBox control, 617
- viewport declaration
 - about, 200
 - block elements and, 228
 - CSS and, 211
 - defining size through meta tags, 203–206
 - device width in, 206–207
 - landscape behavior, 208–210
 - metadata attributes, 205
 - for older devices, 214
 - pixel availability, 208
 - problems using, 200–203
 - usage compatibility table, 213
- Viewporter mini-framework, 207
- viewport_supported capability (WURFL), 318
- Virtual Developer Labs (VDLs), 121
- visibility property (CSS), 439
- visibilitychange event, 457
- visited pseudoclass (CSS), 413
- Visual Basic language, 308
- Visual Studio (Microsoft), 91
- VMWare Player, 107
- Vorbis audio files, 379

W

- W3C
 - animation timing specification, 475
 - APIs for mobile web apps, 594
 - Contacts API, 649
 - Device API Working Group, 594

- DeviceOrientation Event specification, 579–583
- DOM specification, 463
- File API, 649
- FileSystem API, 520
- Geolocation API, 534–538
- IndexedDB API, 515, 518–519
- Mobile Accessibility website, 232
- mobile markup checker, 677
- Mobile Web Initiative, 177
- Network Information API, 649
- performance APIs, 688
- Pointer Events specification, 569–572
- Touch Event API, 568
- Web Workers specification, 464–466
- Widget Interface specification, 636
- Widget Packaging and Configuration standard, 601, 624
- WAI-ARIA (Accessible Rich Internet Applications), 232, 232
- WAP (Wireless Application Protocol)
 - history of, 39–43
 - HTTP and, 300
 - WTAI and, 553
- WAP CSS (WCSS)
 - about, 172
 - extensions for, 172–176
 - validating forms, 275–277
- WAP Link, 41
- WAP Push, 41
- wap subdomain, 41
- wap-accesskey attribute, 173
- wap-input-format attribute, 176, 275
- wap-input-required attribute, 176, 275
- wap-marquee attribute, 174
- wap-marquee-dir attribute, 174
- wap-marquee-loop attribute, 174, 174
- wap-marquee-style attribute, 174
- wap_push capability group (WURFL), 317
- WAV format, 379
- WBMP (Wireless Bitmap), 158
- WCSS (WAP CSS)
 - about, 172
 - extensions for, 172–176
 - validating forms, 275–277
- web apps, 501
 - (see also native web apps)
 - architecture of, 599–601
 - BlackBerry support, 624–632
 - creating for iOS, 602–612
 - full screen apps patterns, 652–654
 - Google Chrome support, 624
 - Mozilla OWA support, 619–623
 - Nokia support for, 632–643
 - offline, 501–511
 - pros and cons, 598–599
 - reloading, 653
 - Samsung support, 624
 - server-side adaptation mechanism, 132
 - standards for, 601–602
 - Symbian support, 612, 637–643
 - web shortcuts and, 702
- Web Audio API, 380
- Web Bluetooth API, 595
- Web Clip icon, 187, 189
- web engines, 49
- web hosting, 125
- Web Intents API, 594, 595
- Web Notifications API, 588–590
- web platforms
 - about, 64
 - audio compatibility table for, 381
 - canvas compatibility table, 364
 - client-side detection, 298–299
 - e-books, 72
 - HTML5 web apps, 64–66
 - image format compatibility table, 336
 - informational websites for, 284
 - native web apps and, 69–72
 - navigator object properties compatibility table, 298
 - pseudo-browsers, 68
 - video compatibility table for, 378
 - web views, 67–67
- Web Runtime (WRT), 637
- web shortcuts, 702
- Web Sockets API, 527
- Web Sockets protocol, 527
- Web SQL Database API, 514–517
- Web Storage API, 511–514
- web views, 67–67, 307
- Web Workers specification, 464–466
- WebAPI initiative (Mozilla), 594
- .webapp file extension, 620
- WebApp.Net library, 487
- webapps (see native web apps)
- WebFM API, 595
- WebGL API, 362

- WebKit layout engine, 49
- webkitBackingStorePixelRatio property, 361
- WebM container, 369
- WebMatrix (Microsoft), 91
- WebNFC API, 595
- webOS platform
 - about, 29
 - emulator for, 107
 - preinstalled browser, 78
- WebP format, 332, 336
- WebPlatform.org website, 284
- WebRTC API, 595
- website icons, 183–186
- websites
 - desktop, 36
 - informational, 281–284
- WebSMS API, 594
- WebTelephony API, 594
- WebWorks engine (BlackBerry)
 - about, 624
 - bbUI.js framework, 628–629
 - configuration file, 624–626
 - packaging and signing, 629
 - Ripple tool, 631
 - useful extension APIs, 626
- Weinre tool, 116, 683
- wgt package, 633
- .wgz file extension, 639
- WHATWG organization, 169, 269
- Widget Interface specification, 636
- widget object (Symbian webapps)
 - isrotationsupported property, 641
 - onexit property, 641
 - onhide property, 641
 - onshow property, 641
 - openApplication() method, 641
 - openURL() method, 641
 - performTransition() method, 641
 - preferenceForKey() method, 641
 - prepareForTransition() method, 641
 - setDisplayLandscape() method, 641
 - setDisplayPortrait() method, 641
 - setNavigationEnabled() method, 641
 - setNavigationType() method, 641
 - setPreferenceForKey() method, 641
- Widget Packaging and Configuration standard, 601, 624
- width attribute
 - CSS3 media queries, 387
 - img element, 336
 - viewports, 205, 211
- WiFi Information API, 595
- WiFi Positioning System (WPS), 60, 531
- Wikimedia website, 61
- Wikipedia website, 129
- window object
 - cancelAnimationFrame() method, 475
 - clearInterval() method, 473
 - close() method, 471
 - onLine property, 584
 - open() method, 471
 - openDatabase() method, 515, 517
 - orientation property, 480, 583
 - requestAnimationFrame() method, 473
 - scrollTo() method, 472
 - scrollY() method, 498
 - setInterval() method, 473
 - setTimeout() method, 473
- windows
 - focus and scroll management, 472
 - JavaScript dialogs, 467–469
 - manipulating, 471
 - pop-up, 416, 471
- Windows 8 platform
 - about, 18
 - native apps support, 613–619
 - URI association from native apps, 563
 - user agent tricks, 307
- Windows Embedded Compact, 19
- Windows Mobile platform, 19
- Windows Phone platform
 - about, 17
 - application and games delivery, 666
 - Bing Maps application, 545
 - emulators for, 107–109
 - Nokia devices and, 20
 - URI association from native apps, 563
- Windows platform
 - application and games delivery, 666
 - brands and models in, 17
 - Start tile, 195–198
- Windows Simulator, 109
- Windows Store App, 223
- WinJS library, 615
- Wireless Application Protocol (WAP)
 - history of, 39–43
 - HTTP and, 300
 - WTAI and, 553

- Wireless Bitmap (WBMP), 158
- Wireless Markup Language (WML), 40, 154
- Wireless Telephony Application Interface (WTAI), 553, 559
- Wireless Universal Resource File (see WURFL)
- WML (Wireless Markup Language), 40, 154
- WMLScript language, 158
- WMV container, 368
- wp library
 - ap function, 559
 - mc function, 555
 - sd function, 556
- WPS (WiFi Positioning System), 60, 531
- wrap attribute, 248
- Wroblewski, Luke, 139, 140, 145
- WRT (Web Runtime), 637
- wta capability group (WURFL), 316
- WTAI (Wireless Telephony Application Interface), 553, 559
- WTF Mobile Web website, 149
- WURFL (Wireless Universal Resource File)
 - about, 313–315
 - APIs supported, 319–323
 - architecture, 315
 - capabilities, 316–318
 - emoji property, 341
 - linking to video files, 370
 - SMS URI schemes and, 557
- WURFL Cloud Service, 313, 322
- WURFLManager class
 - getCapabilitiesNameForGroup() method, 321
 - getDevice() method, 321
 - getDeviceForHttpRequest() method, 321
 - getDeviceForUserAgent() method, 321
 - getListOfGroups() method, 321
- WURFL_Device class
 - getAllCapabilities() method, 321
 - getCapability() method, 321
- WYSIWYML editor, 253

X

- X-Callback-URL specification, 562
- X-Forwarded-For header (HTTP), 310, 311, 540
- X-Icon Editor, 183
- X-Mobile-Gateway header (HTTP), 311
- X-Nokia-Device-Type header (HTTP), 311
- x-nokia-musicshop-bearer header (HTTP), 308
- X-OperaMini-Features header (HTTP), 310

- X-OperaMini-Phone header (HTTP), 310
- X-OperaMini-Phone-UA header (HTTP), 310
- X-WAP-Profile header (HTTP), 306, 311
- x-webkit-airplay attribute, 375
- xhr2 property, 523
- XHTML
 - about, 164
 - available tags, 165
 - creating compatible templates, 167
 - DOCTYPE for, 161
 - embedding SVG in, 354
 - i-mode, 230
 - markup additions, 169
 - official noncompatible features, 166
 - preferred MIME type, 161
- XHTML.MP (Mobile Profile)
 - about, 164
 - available tags, 165
 - closing elements, 183
 - creating compatible templates, 167
 - DOCTYPE for, 161
 - inputmode attribute, 269
 - markup additions, 169
 - official noncompatible features, 166
 - preferred MIME type, 161
 - WAP and, 41
- xhtml_file_upload capability (WURFL), 318
- xhtml_format_as_css_property capability (WURFL), 318
- xhtml_make_phone_call_string capability (WURFL), 318
- xhtml_send_sms_string capability (WURFL), 318
- xhtml_supports_iframe capability (WURFL), 318
- xhtml_support_level capability (WURFL), 318
- xhtml_ui capability group (WURFL), 316
- XMLHttpRequest object, 523, 523
 - (see also Ajax)
- Xoom tablet, 16, 28
- Xperia series (Sony Mobile), 27, 55
- Xpress Nokia Browser, 77, 310
- Xpress Web App Builder for Series 40, 702
- XUI framework
 - about, 486
 - addClass() method, 487
 - css() method, 487
 - getStyle() method, 487
 - html() method, 486

- on() method, 487
- removeClass() method, 487
- setStyle() method, 487
- tween() method, 487
- xhr() function, 487
- xhrjson() function, 487

Y

- Yahoo! PlaceMaker API, 533
- Yahoo! search engine, 695
- Yahoo! Web Analytics, 705

- YouTube
 - Android intents and, 562
 - iOS URL schemes, 560

Z

- Zakas, Nicholas, 693
- Zepto.js library, 485, 573
- ZIP files, 601
- zoom attribute, 212
- zoom experience (browsers), 45
- Zuckerberg, Mark, 70

About the Author

Maximiliano Firtman, @firt, is a developer focused on mobile and HTML5 development. He is a trainer in mobile technologies and founder of ITMaster Professional Training. He is the author of many books, including *Programming the Mobile Web* (O'Reilly). He has spoken at international conferences such as OSCON, Velocity, Breaking Development, GOTO Europe, Campus Party, QCon, and Adobe en Vivo.

He has been an Adobe Community Professional since 2011, a BlackBerry Elite Developer since 2013, and a Nokia Developer Champion since 2006, and he has developed many mobile-related projects, such as *MobileHTML5.org*, *MobileTinyURL.com*, and *iWebInspector.com*. He maintains a mobile web development blog at *MobileXweb.com*.

He is an expert in native and HTML5 web development, including iOS, Android, PhoneGap, and jQuery technologies.

Colophon

The animal on the cover of *Programming the Mobile Web, Second Edition* is a jerboa, a small jumping rodent of the family Dipodidae. The 33 species of jerboa are found in deserts of Asia and North Africa. They feed on the leaves and roots of desert plants; many species also eat insects. They extract water from their food so efficiently that they do not need to drink.

Jerboas' powerful hind legs may be four times longer than their front legs and enable them to hop up to three meters. Their tails, which are often tufted, are longer than their bodies and are used for balance. Their ears vary from species to species—they may be small and mouselike or broad and rabbitlike.

Jerboas are well adapted to their harsh desert environments. They are nocturnal and hide in burrows, which they may plug for protection against the elements, during the day. Some jerboas living in hot regions enter a state of torpor (estivation) during the hottest months; jerboas living in cold regions hibernate during the winters.

The cover image is from Riverside Natural History. The cover font is Adobe ITC Garamond. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.

Have it your way.

O'Reilly eBooks



- Lifetime access to the book when you buy through oreilly.com
- Provided in up to four DRM-free file formats, for use on the devices of your choice: PDF, .epub, Kindle-compatible .mobi, and Android .apk
- Fully searchable, with copy-and-paste and print functionality
- Alerts when files are updated with corrections and additions

oreilly.com/ebooks/

Safari Books Online

- Access the contents and quickly search over 7000 books on technology, business, and certification guides
- Learn from expert video tutorials, and explore thousands of hours of video on technology and design topics
- Download whole books or chapters in PDF format, at no extra cost, to print or read on the go
- Get early access to books as they're being written
- Interact directly with authors of upcoming books
- Save up to 35% on O'Reilly print books



See the complete Safari Library at safari.oreilly.com

O'REILLY®

Spreading the knowledge of innovators.

oreilly.com

Get even more for your money.

Join the O'Reilly Community, and register the O'Reilly books you own. It's free, and you'll get:

- \$4.99 ebook upgrade offer
- 40% upgrade offer on O'Reilly print books
- Membership discounts on books and events
- Free lifetime updates to ebooks and videos
- Multiple ebook formats, DRM FREE
- Participation in the O'Reilly community
- Newsletters
- Account management
- 100% Satisfaction Guarantee

Signing up is easy:

- 1. Go to: oreilly.com/go/register**
- 2. Create an O'Reilly login.**
- 3. Provide your address.**
- 4. Register your books.**

Note: English-language books only

To order books online:

oreilly.com/store

For questions about products or an order:

orders@oreilly.com

To sign up to get topic-specific email announcements and/or news about upcoming books, conferences, special offers, and new technologies:

elists@oreilly.com

For technical questions about book content:

booktech@oreilly.com

To submit new book proposals to our editors:

proposals@oreilly.com

O'Reilly books are available in multiple DRM-free ebook formats. For more information:

oreilly.com/ebooks

O'REILLY®

Spreading the knowledge of innovators

oreilly.com

©2015 O'Reilly Media, Inc. O'Reilly logo is a registered trademark of O'Reilly Media, Inc. 00000

Programming the Mobile Web

With the second edition of this popular book, you'll learn how to build HTML5 and CSS3-based apps that access geolocation, accelerometer, multi-touch screens, offline storage, and other features in today's smartphones, tablets, and feature phones. The market for mobile apps continues to evolve at a breakneck pace, and this book is the most complete reference available for the mobile web.

Author and mobile development expert Maximiliano Firtman shows you how to develop a standard app core that you can extend to work with specific devices. This updated edition covers many recent advances in mobile development, including responsive web design techniques, offline storage, mobile design patterns, and new mobile browsers, platforms, and hardware APIs.

- Learn the particulars and pitfalls of building mobile websites and apps with HTML5, CSS, JavaScript, and responsive techniques
- Create effective user interfaces for touch devices and different resolution displays
- Understand variations among iOS, Android, Windows Phone, BlackBerry, Firefox OS, and other mobile platforms
- Bypass the browser to create native web apps, ebooks, and PhoneGap applications
- Build apps for browsers and online retailers such as the App Store, Google Play Store, Windows Store, and App World

Maximiliano Firtman, Founder of ITMaster Professional Training and an educator on web and mobile technologies, is an expert in native and HTML5 mobile web development, including iOS, Android, BlackBerry, and Windows Phone. He's the author of several technical articles and books, including *jQuery Mobile: Up and Running* (O'Reilly).

"Programming the Mobile Web will not only give you a quick start for any new platform you're looking for, it also lets you see the big picture of this flourishing technology."

—Carlos Solis
Founder, *Revolucion.mobi*



Twitter: @oreillymedia
facebook.com/oreilly

O'REILLY®
oreilly.com