THE MODERN WEB

MULTI-DEVICE WEB DEVELOPMENT WITH HTML5, CSS3, AND JAVASCRIPT

PETER GASSTON

6

DEVICE APIS

In the previous chapters, I've discussed some of the many APIs that have been introduced as part of the HTML5 process, such as microdata and Touch Events. But there is a further range of APIs that, although not part of the spec, are certainly related; and these APIs offer something extremely attractive to developers in the multi-screen world: access to the device itself.

In this chapter, we take a look at some device APIs—from the new location and spatial features in portable devices to file and storage options across most modern browsers. Obviously not all APIs are going to be available on every device—knowing the position in three-dimensional (3-D) space of a television is of little practical use—but many APIs are useful across a broad range of user agents.

This is a curated list of those APIs I feel will be most practical, and the introductions to many are, for reasons of space, quite brief; often, the APIs will be much more extensive, and although I'll note where I think the scope is available for you to learn more, I urge you to discover for yourself the capabilities and possibilities of accessing the device through JavaScript.

NOTE: *The examples and demos in this chapter are interactive; I've included screenshots and illustrations in some cases, but if there's one chapter you really should download the example files for, it's this one.*

Geolocation

Location-based services are handy in all sorts of ways, from helping users with mobile devices find their way around to providing tailored information about the region they live in. The Geolocation API accesses a device's location services, which use GPS, wireless, or cell tower data to provide information about the device's location that will be used by your location-based apps.

Location data obviously involves privacy concerns, so in most (if not all) browsers, users must give explicit permission to access this data, usually in the form of an on-screen prompt that allows them to opt in or out of providing their location, as shown in Figure 6-1.
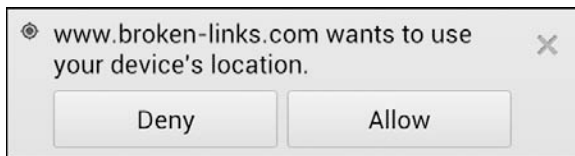

Figure 6-1: The Geolocation opt-in prompt in Chrome for Android

The data is held in the `geolocation` object, a child of `window.navigator`, which you can access using the `getCurrentPosition()` method:

```
navigator.geolocation.getCurrentPosition(function(where){
  // Do something
});
```

A successful callback returns an object (I've called it *where*) containing a `coords` child object. This child object has a series of properties pertaining to the user's position, such as his or her altitude and heading, but the ones I'm really interested in are `latitude` and `longitude`. In this code, I'm accessing these properties and displaying them in an alert:

```
navigator.geolocation.getCurrentPosition(function(where){
  alert(where.coords.latitude + ',' + where.coords.longitude);
});
```

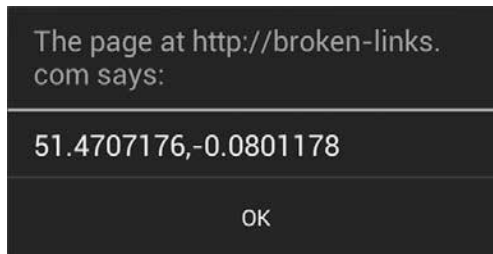You can try this for yourself in *position-current.html*; my results are shown in Figure 6-2.

Figure 6-2: Coordinates obtained through the
geolocation object, referencing the street I live on;
please don't stalk me.

Occasionally an error is returned when looking for the position; an optional error callback can check for this. The following code creates two functions, one for successful location and one for an error:

```
var geo = navigator.geolocation,
    lcn_success = function(where) { ... },
    lcn_error = function() { ... };
geo.getCurrentPosition(lcn_success, lcn_error);
```

Sometimes a GPS device can take a little while to find the user's exact position, and, of course, the user may also be on the move, so instead of a one-off location, you can choose to watch the user's position, receiving updated results when location data changes. You do this with the watchPosition() method, also on the geolocation object, which works in the same way as getCurrentPosition():

```
navigator.geolocation.watchPosition(function(where){
  console.log(where.coords.latitude,where.coords.longitude);
});
```

To cancel watching a user's position, use the clearWatch() method with the unique ID created by the watchPosition() method; in this example, the process ends when the user clicks the #*stop* link:

```
var geo = navigator.geolocation,
    watchID = geo.watchPosition( ... ),
    endWatch = document.getElementById('stop');
endWatch.addEventListener('click', function () {
  geo.clearWatch(watchID);
}, false);
```

In *position-watch-clear.html*, you can see a demo of this in action. Open the page in a mobile device and move around, and you should see the location update as your device gets a better fix on your location.

Orientation

The Orientation API detects changes to the device's position in 3-D space— that is, movement up and down, left and right, and clockwise and counter-clockwise. This movement is measured with an accelerometer, and the devices that are most likely to contain one are those that are most portable; mobile phones and tablets move frequently so are very likely to have one, laptops move to some degree so may contain one, and desktops and TVs move so infrequently that it's very unlikely they'll have an accelerometer or access to this API.

Using orientation events opens up new possibilities for interaction and navigation; some apps already provide an option to control page scrolling by tilting the device forward or backward, and navigation between tiles or pages by tilting to the left or right.

Before detailing the API, I should talk about three-dimensional axes (you can skip this paragraph if you know about them already). All movement in three dimensions has three directions, or *axes*, commonly referred to as *x*, *y*, and *z*. If you hold a device in front of you now (or imagine you are doing so), the *x*-axis runs from left to right, *y* from top to bottom, and *z* toward you and away from you, as shown in Figure 6-3. Movement is measured along these axes from the center of the device and is either positive or negative: Bringing the device closer to you moves it positively along the *z*-axis and away moves it negatively. Lowering the device toward your feet moves it negatively along the *y*-axis and moving it to your right moves it positively along the *x*-axis.
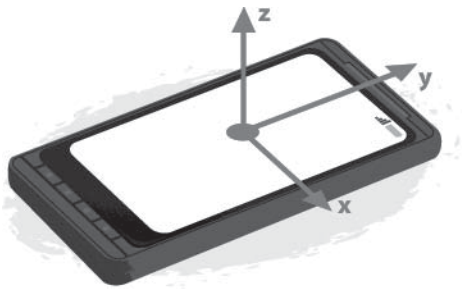

Figure 6-3: Movement along the three-dimensional axes (This image is taken from the Mozilla Developer Network [MDN] article, "Orientation and Motion Data Explained": http://developer.mozilla.org/en-US/docs/DOM/Orientation _and_motion_data_explained/. It is used under a Creative Commons license.)

To detect the movement along each axis, use the `deviceorientation` event on the `window` object. This event fires every time the device moves and returns an object with a series of useful orientation properties:

```
window.addEventListener('deviceorientation',function (orientData) {
  ...
}, false);
```

The three key properties that are germane to movement are `alpha`, `beta`, and `gamma`. Each is measured with a number representing degrees of rotation, although some are constrained within set limits.

- `alpha` measures rotation around, not movement along, the *z*-axis—that is, if the device were laid flat on a table, clockwise or counterclockwise movement. The value of `alpha` is a number from 0 to 360.

- `beta` is rotation around the *x*-axis, which you can picture as tipping the top edge of the device toward or away from you. `beta` has a value range of –180 (tip toward you) to 180 (tip away from you).

- `gamma` is rotation around the *y*-axis or tilting the device from side to side. The value of `gamma` ranges from –90 (tip left) to 90 (tip right).

As a very simple example, the code in the following script uses `deviceorientation` to detect changes to the orientation and then logs the three values to the console:

```
window.addEventListener('deviceorientation',function (orientData) {
  console.log(orientData.alpha,orientData.beta,orientData.gamma);
}, false);
```

In the example file *orientation.html*, you can see a slightly different version that updates the text on the page when orientation changes; open it on a mobile or tablet device and move the device around to see the page content update.

Fullscreen

We all know the Web is an immensely powerful distraction machine, so sometimes providing an option to focus only on the content at hand is useful. This functionality is provided by the Fullscreen API, which allows you to expand any element to fill the entire screen of the device, rather than just the browser viewport. This is especially handy for large-screen devices, for instance, when playing video to provide the "lean back" experience of television.

Before setting up this script, check whether the browser has a fullscreen mode. You can do this with the Boolean `fullScreenEnabled` attribute:

```
if (document.fullScreenEnabled) { ... }
```

Fullscreen mode is called with the `requestFullScreen()` method. As this introduces potential security risks (an often-quoted example is an attack website that fools you into thinking that you're seeing something else and copies your keystrokes), many devices provide an on-screen prompt to make sure you give permission to enter fullscreen mode. If you grant that permission, the element the method is called on scales up to 100 percent of the device screen's height and width.

In the next code snippet, a click event listener is applied to the element *#trigger*, which, when fired, will put *.target* into fullscreen mode, as long as permission is granted. You can see this for yourself in the file *fullscreen.html*, which is illustrated in Figure 6-4.

```
var el = document.querySelector('.target'),
    launch = document.getElementById('trigger');
launch.addEventListener('click', function () {
  el.requestFullScreen();
}, false);
```


Figure 6-4: An element launched into fullscreen mode with an on-screen alert in Firefox for Android

The browser should offer a means to exit fullscreen mode, but you can also provide your own with the exitFullScreen() method. The next code block shows a function that uses this method to leave fullscreen mode when the ENTER key is pressed. Note two further things in the code: First, it uses the fullscreenchange event, which is fired whenever an element enters or leaves fullscreen mode; and second, it relies on an if statement using the fullScreenElement attribute, which returns either information about the element that is in fullscreen mode or null if there is none.

```
document.addEventListener('fullscreenchange', function () {
  if (document.fullScreenElement !== null) {
    document.addEventListener('keydown', function (e) {
      if (e.keyCode === 13) {
        document.exitFullScreen();
      }
    }, false);
  }
}, false);
```

When an element has been put in fullscreen mode, you might want to style it (or its children) a little differently. It's proposed that you can do this with a new dedicated CSS pseudo-class, which will be called either `:fullscreen` or `:full-screen`:

```
.target:full-screen {}
```

Vibration

The Vibration API makes a device vibrate, providing some haptic feedback for your users. This API actually used to be called the Vibrator API, but I'm sure you don't need me to tell you why that name was changed very quickly. Obviously not all devices are capable of vibrating, especially larger ones, so this API is decidedly more useful in mobile devices.

The API is extremely simple, requiring only the `vibrate()` method on the `navigator` object. The value supplied to `vibrate()` is a figure representing the number of milliseconds for the device to vibrate; for example, to make the device vibrate for one-fifth of a second after the user has completed a `touchend` event, use this code:

```
document.addEventListener('touchend', function () {
  window.navigator.vibrate(200);
});
```

You can also use an array of values that alternate between vibrations and pauses; that is, the odd-numbered values are vibrations and the even values are pauses. In this example, the device vibrates for 200ms, pauses for 200ms, and then vibrates for 500ms:

```
document.addEventListener('touchend', function () {
  window.navigator.vibrate([200,200,500]);
});
```

Vibrating runs down the battery more quickly, so use this API with caution. You can manually stop a vibration by using a 0 or an empty array value. In this code, the device will begin to vibrate for 5 seconds when the touch event starts, and then stops when the event ends:

```
document.addEventListener('touchstart', function () {
  window.navigator.vibrate(5000);
});
document.addEventListener('touchend', function () {
  window.navigator.vibrate(0);
});
```

You can try the API for yourself in the example file *vibration.html*, even though obviously you'll need to open it on a mobile device with vibration capabilities if you want to actually feel the vibrations. If you don't have one on hand, Figure 6-5 shows a reconstruction of the experience.

Figure 6-5: The Vibration API in action (reconstruction)

Battery Status

One of the key concerns with portable devices is knowing their battery status. Mobile devices can get as little as seven or eight hours out of a full charge, whereas a laptop is lucky to get more than three or four hours. Knowing the status of the device's battery can be important before you begin power-hungry processes or commence to download large files.

You can get information about the battery with the Battery Status API, which brings a set of attributes on the navigator.battery object. For example, to find out if the battery is currently charging, you can use the charging attribute to get a true or false value:

```
var batteryStatus = navigator.battery.charging;
```

To find the current battery level, you can use the level attribute, which returns a value from 0 (empty) to 1 (fully charged). The following code is a simple demonstration of this in action: The battery level is obtained and its value used as the value of a meter element (which will be fully introduced in Chapter 8), and the current charging status ('*Charging*' or '*Discharging*') is appended below it. You can try it yourself in the example file *battery.html*. The result is shown in Figure 6-6.

Figure 6-6: A meter element showing the remaining battery level of my device, plus its charging status

```
var el = document.getElementById('status'),
    meter = document.querySelector('meter'),
    battery = navigator.battery,
    status = (battery.charging) ? 'Charging' : 'Discharging';
meter.value = battery.level;
meter.textContent = battery.level;
el.textContent = status;
```

The battery object has two further attributes: chargingTime and dischargingTime. Both of these return a value, in seconds, of the remaining time until the battery is fully charged or fully discharged, respectively.

The Battery Status API also has a series of events that fire when a change to any of the attributes is detected: `chargingchange`, `chargingtimechange`, `dischargingtimechange`, and `levelchange`. The following code uses `chargingchange` to detect a change to the device's charging status and fires an alert if the status has changed:

```
var status,
    battery = navigator.battery,
    chargeStatus = function () {
      (battery.charging) ? status = 'Charging' : status = 'Discharging';
      return status;
};
battery.addEventListener('chargingchange', function () {
  window.alert(chargeStatus());
}, false);
window.alert(chargeStatus());
```

You can try this one yourself using the example file *battery-event.html*—plug and unplug your phone from its charger to see the status update.

Network Information

Knowing the current strength of a device's Internet connection is extremely useful; you may want to serve lower-resolution images to devices with low bandwidth or stream different video qualities to users depending on their connection. Likewise, you may want to hold off on the background processes if the user has a limited or metered tariff.

The Network Information API is composed of two attributes on the `connection` object: `bandwidth`, which is a figure representing the *estimated* bandwidth in Megabytes (MBps) of the current connection (0 if the device is offline, `infinity` if the result is unknown); and `metered`, a Boolean that returns `true` if the connection is metered (such as on pay-as-you-go tariffs).

The following code shows a function that uses both attributes: `bandwidth` to return the current connection's bandwidth and `metered` to add an extra message to the status if the connection is limited.

```
var status,
    connection = navigator.connection,
    showStatus = function () {
      status = connection.bandwidth + ' MB/s';
      if (connection.metered) {
        status += ' (metered)';
      }
        alert(status);
    };
showStatus();
```

Network Information also has an event handler, `change`, on the `connection` object, which fires whenever the connection status changes; with this, you can easily add an extra call to the function when necessary:

```
connection.addEventListener('change', showStatus, false);
```

You can see both at work in the file *network.html*—try connecting or disconnecting your Wi-Fi service to see the `change` event fire.

Camera and Microphone

Cameras and microphones have been common on desktop and laptop computers for a long time, and with the rise of mobile devices they've become extremely prevalent—almost ubiquitous. But for years, we've had to rely on third-party plug-ins, such as Flash and Java, to get audio and video input on the Web, so a native input method is more than overdue.

This native input comes in the shape of the `getUserMedia()` method, part of the WebRTC project, which I'll discuss in more detail in Chapter 9. The `getUserMedia()` method is on the `navigator` object, and takes up to three arguments: The first is for options about the stream, such as whether to accept only audio, only video, or both; the second is a callback fired when a successful connection is made; and the third, which is optional, is a failure callback:

```
navigator.getUserMedia({options}, success, failure);
```

As with the Geolocation and Fullscreen APIs, accessing the user's camera or microphone has privacy implications, so many browsers provide an on-screen prompt asking for the user's permission to access the device. On devices with more than one camera, some user agents offer a native control to switch between them.

A media stream requires a special element in order to be displayed, either the new `video` or `audio` HTML5 element (depending on the stream content). I introduce these new elements fully in Chapter 9, but for the purposes of the following demonstration, using a video stream, you need the following markup somewhere on your page:

```
<video autoplay></video>
```

When the successful callback is fired from `getUserMedia()`, the media stream is returned with a unique ID (provided by you), which will be supplied to the `video` element. The following code shows a basic example, which I've annotated and will explain after:

```
❶ navigator.getUserMedia({video:true}, function (stream) {
❷   var video = document.querySelector('video');
❸   video.src = window.URL.createObjectURL(stream);
  });
```

In line ❶, I've supplied two arguments to the `getUserMedia()` method: The first is the stream options where I'm flagging that I want to get video, no audio; and the second is the callback function where I've given the result a unique ID of *stream*. In the next line ❷, I've used `querySelector()` to assign the `video` element to the *video* variable so that in line ❸, I can use the `createObjectURL()` method to convert *stream* into a URL and set it as the `src` attribute of the `video` element. No failure callback is supplied.

To try this for yourself, see the file *getusermedia.html*—you'll need to have a video on your device to see the file in action.

Web Storage

Recording information about previous activity is usually done with cookies, but one of their drawbacks is that you can store only small amounts of data. The Web Storage API was created to allow user agents to store more data on the user's device. This data can be stored only until the browser is closed, which is known as *session storage*, or kept until the user or another script actively flushes the data, which is called *local storage*. Both operate in essentially the same way, except for that one key difference—permanence.

To store data, you save it in key:value pairs, similar to how you store cookies now, except the quantity of data that can be saved is greater. The API has two key objects, which are straightforward and memorable: `localStorage` for local storage and `sessionStorage` for session storage.

N O T E :   *In the examples in this section I use* `sessionStorage`, *but you can swap this for* `localStorage` *if you prefer more permanent storage; the syntax applies equally.*

The web storage syntax is pretty flexible, allowing three different ways to store an item: with the `setItem()` method, with square bracket notation, or with dot notation. As a simple example, the next code snippet shows how you might store this author's name; all three different ways of storing data are shown for comparison, and all are perfectly valid.

```
sessionStorage.setItem('author','Peter Gasston');
sessionStorage['author'] = 'Peter Gasston';
sessionStorage.author = 'Peter Gasston';
```

Some developer tools allow you to inspect the contents of storage, so Figure 6-7 shows the result of this code, regardless of which approach you use.

Retrieving items from storage is just as flexible a process; you can use the `getItem()` method, which accepts only the name of the relevant key as an argument, or the square bracket or dot notation method without any value. In the next code snippet, all three techniques are shown and are equivalent:

```
var author = sessionStorage.getItem('author');
var author = sessionStorage['author'];
var author = sessionStorage.author;
```

Figure 6-7: A key:value pair stored in the browser, shown in the
WebKit Web Inspector

Although I'm storing only very simple values in these examples, in most browsers, you can store up to 5MB of data for each subdomain. This is the figure recommended in the specification, although it's not mandatory.

You can delete a single item from storage using the removeItem() method, which like getItem(), takes a single key name as an argument and deletes the stored item with the matching key:

```
sessionStorage.removeItem('author');
```

In the file *storage.html*, I've put together a simple demo that adds and removes items from the storage. To see the result, you need developer tools that show the contents of the storage, such as in the Resources tab of the WebKit Web Inspector. The contents don't update in real time, so you have to refresh to see changes.

The nuclear option to remove all items in storage (although only on the specific domain storing them, of course) is the clear() method:

```
sessionStorage.clear();
```

A storage event on localStorage is fired whenever storage is changed. This returns an object with some useful properties such as key, which gives the name of the key that has changed, and oldValue and newValue, which give the old and new values of the item that has changed. Note this event fires only on other open instances (tabs or windows) of the same domain, not the active one; its utility lies in monitoring changes if the user has multiple tabs open, for example.

The next code block runs a function that fires whenever storage is modified and logs an entry into the console. You can try it yourself in the file *storage-event.html*, but you'll need to open the file and developer console in two different tabs to see the changes occur—remember, changes to the value will show in the other window, not the one where the click occurs.

```
window.addEventListener('storage', function (e) {
  var msg = 'Key ' + e.key + ' changed from ' + e.oldValue + ' to ' + e.newValue;
  console.log(msg);
}, false);
```

Storage is being taken even further with the development of the Indexed Database (IndexedDB) API, which aims to create a full-fledged storage database in the browser that you access via JavaScript. Many browsers have already made an attempt at this, but the vendors couldn't decide on a common format. IndexedDB is an independently created standard aimed at keeping everyone happy. Its heavily technical nature takes it out of the scope of this book, but if you need advanced storage capabilities, keep it in mind.

Drag and Drop

Adding a "physical" aspect to your websites that allows users to move elements around the screen is a nice option. This "drag and drop" behavior is especially useful on devices with touch interfaces.

The Drag and Drop API is probably the oldest feature I cover in this book. It was first implemented in Internet Explorer 5 back in 1999 (that's about 150 Internet years ago) and has been adopted by other browsers for quite some time, although the effort to standardize it was only undertaken as part of the HTML5 movement. Unfortunately, Drag and Drop shows some signs of aging, being quite arcane and unintuitive at first.

By default the a and img elements can be dragged around the screen (I'll get to other elements momentarily), but you have to set up a *drop zone*, an area that the elements can be dragged into. A drop zone is created when you attach two events to an element: dragover and drop. All that's required of dragover is that you cancel its default behavior (for one of the arcane reasons I noted earlier, which you don't need to worry about). All the hard work happens with the drop event.

That may sound a little confusing, so this example shows a very simple setup: The #*target* element has the dragover and drop event listeners attached to it, the callback function of dragover prevents the default behavior with preventDefault(), and the main action happens inside the callback function of the drop event.

```
var target = document.getElementById('target');
target.addEventListener('dragover', function (e) {
  e.preventDefault();
}, false);
target.addEventListener('drop', function (e) {
  // Do something
}, false);
```

All of the events in the Drag and Drop API create an object called `dataTransfer`, which has a series of relevant properties and methods. You want to access these when the `drop` event is fired. For img elements, you want to get the URL of the item, so for this you use the `getData()` method with a value of *URL* and then do something with it; in this example, I'll create a new img element and pass the URL to the `src` attribute, making a copy of the existing one:

```
target.addEventListener('drop', function (e) {
  e.preventDefault();
  var newImg = document.createElement('img');
  newImg.setAttribute('src', e.dataTransfer.getData('URL'));
  e.currentTarget.appendChild(newImg);
}, false);
```

Note the use of `preventDefault()` again inside the function on the `drop` callback; using this is important, because in most (if not all) browsers the default behavior after dropping an item into a drop zone is to try to open its URL. This is part of Drag and Drop's arcane behavior. All you really need to know is to use `preventDefault()` to stop this from happening.

You can see a simple example based on the previous code in the file *drag-drop.html*—just drag the image from its starting position to inside the box.

I said previously that, by default, only a and img elements are draggable, but you can make that true of any element in two steps. First, apply the `true` value to the `draggable` attribute of the element in question:

```
<div draggable="true" id="text">Drag Me</div>
```

Second, specify a datatype for the element. You do this with the `dragstart` event, using the `setData()` method of `dataTransfer` to apply a MIME type (in this case, `text/plain`) and a value (in this case, the text content of the element):

```
var txt = document.getElementById('txt');
  txt.addEventListener('dragstart', function (e) {
  e.dataTransfer.setData('text/plain', e.currentTarget.textContent);
}, false);
```

You can detect the type of file being dropped by using the `contains()` method, which is a child of the `types` object, itself a child of the `dataTransfer` object created in the callback function of the `drop` event. The method returns `true` or `false` if the string supplied in the argument matches a value in `types`; for example, to find out if a dropped element contains a plain text type, you would use this:

```
var foo = e.dataTransfer.types.contains('text/plain');
```

Using the `contains()` method means you can perform different actions on different files.

The example file *drag-drop-2.html* shows two elements, an `img` and a `p`, which can be dragged into the marked drop zone, creating a copy of each, and the following code shows how this is done: The `contains()` method detects if the element being dragged contains a URL; if it does, it must be an `img`, so it creates a new `img` element with the URL of the dropped element in the `src` attribute; if it doesn't, it must be text, so it creates a new text node filled with the text of the dropped element.

```
target.addEventListener('drop', function (e) {
  var smth;
  e.preventDefault();
  if (e.dataTransfer.types.contains('text/uri-list')) {
    smth = document.createElement('img');
    smth.setAttribute('src', e.dataTransfer.getData('URL'));
  } else {
    smth = document.createTextNode(e.dataTransfer.getData('Text'));
  }
  e.currentTarget.appendChild(smth);
}, false);
```

Although what I've described in this section is more than sufficient for you to use the Drag and Drop API, the API contains plenty more that I haven't covered. If you're interested, a number of extra events are available: `dragenter` and `dragleave` are events for the drop zone, and `dragend` and `drag` are fired on the draggable item.

Interacting with Files

Working with different files is a common activity—although much more so on desktops or laptops than on mobile devices—so an API is available for doing this on the Web too. The File API is a fairly low-level API that allows you to get information about files and to access their contents, and there are a few higher-level APIs that I'll mention in due course.

To access files, you can either choose them using the `file` input element or drag them from a folder on your system (depending on the system you use) with the Drag and Drop API, which is the approach we'll look at here.

The `dataTransfer` object, which I just discussed in the previous section, contains a `files` child object that contains a list of all the files dropped into the drop zone. Each file has three properties—`name`, `size`, and `type`—and the meaning of these should be pretty obvious.

The following code example shows a function where files dropped into the drop zone will have their names listed. You do this with a `for` loop that runs through the `files` object and outputs the `name` property for each. Try it for yourself with the example file *files.html*.

```
target.addEventListener('drop', function (e) {
  var files = e.dataTransfer.files,
      fileNo = files.length;
  e.preventDefault();
  for (i = 0; i < fileNo; i++) {
    var el = document.createElement('li'),
        smth = document.createTextNode(files[i].name);
    el.appendChild(smth);
    e.currentTarget.appendChild(el);
  }
}, false);
```

If you need more than just information about the file, the FileReader interface allows you to get the content as a text file or data URL (where relevant). The following code snippet shows a simple example using an image file as the source; the syntax is a little complex, so I've annotated the code and will explain it next.

```
    target.addEventListener('drop', function (e) {
      e.preventDefault();
      var files = e.dataTransfer.files[0],
❶      reader = new FileReader();
❷    reader.addEventListener('load', function (evt) {
        var img = document.createElement('img');
❸      img.src = evt.target.result;
        target.appendChild(img);
      }, false);
❹    reader.readAsDataURL(files);
❺    reader.addEventListener('error', function (evt) {
        console.log(evt.target.error.code)
      }, false);
    }, false);
```

In ❶, a new FileReader object is created and assigned to the variable *reader*. To this object, a new event listener is added ❷, which will fire when the file has finished loading, running a function that will create a new img element using the content of the uploaded file. The src for the img element is obtained in ❸, using the result attribute of target, a child object of the event. The type of result is determined in ❹ using the readAsDataURL() method, which encodes the file content as a 64-bit data string. Finally, an error event listener is added to the object in ❺, which uses the code attribute of the error object of the target object of the event object (phew!) to log an error message.

Try this for yourself in *file-2.html*; drag an image from a folder on your system (if possible) to see it appear in the page. In addition to readAsDataURL(), a few other methods are available: readAsText() returns the content of the file as plain text, and readAsArrayBuffer() returns the content as a fixed-length data buffer (especially useful for images).

You can also use a number of APIs to go even further with files: The File Writer API allows you to modify the content of a file, and the File System API goes further still with the provision of a navigable filesystem on the user's device. These APIs are exciting but somewhat too technical for me to go into detail in this book.

Mozilla's Firefox OS and WebAPIs

A potentially quite interesting entry into the mobile OS market comes from Mozilla, makers of the Firefox browser. Mozilla is building a brand new OS from the ground up, all constructed with open web standards—HTML, CSS, JavaScript, and others. The OS has no middleware layer, as iOS or Android does, and the system APIs all use JavaScript to interact directly with the hardware. Building a new OS is a bold undertaking, and I look forward to seeing how it performs.

As part of that effort, Mozilla realized they needed to develop many of the existing device APIs and create many more. This project is called WebAPI, and although many of the included APIs have been covered already in this chapter, a few are unique to Firefox OS at the moment.

Here are some of the new APIs: WebTelephony, for sending and receiving calls; WebSMS, for sending, receiving, and managing text messages; Contacts, for accessing and managing the address book; and Device Storage, for accessing shared files or folders such as the picture gallery featured on many phones.

The fate of Firefox OS and broader implementation of these APIs remain to be seen, but I'm quite excited about a smartphone OS that is built using only open web technologies and the many possibilities that opens up for web developers with regard to device interactions.

PhoneGap and Native Wrappers

If you need deeper access to device APIs but still want to develop using web technologies, you might want to consider a native wrapper for your app. These wrappers act as a kind of layer between your web application and the device in question, providing hooks into the API but not using native code to display what's on the screen. Using a native wrapper around web technologies creates what's known as a *hybrid app*.

Which wrapper you use depends largely on your targets (I'll discuss this in more detail in Chapter 10), but as an example of what they can do, PhoneGap is perfect. It's a wrapper for mobile apps, providing a common API for developers to build hybrid apps that work across iOS, Android, Windows Phone, Blackberry, and more.

Summary

By necessity, I could detail only a few of the many APIs that make up the web platform, including those for location and spatial movement, status of the battery and Internet connection, access to the camera and microphone, local storage capabilities, interaction with files and elements in a tactile way, and access to information about the content of files. I hope that with this overview and the example files I've been able at least to hint at the creative possibilities that open up when you access a device through JavaScript.

Further Reading

Dive Into HTML5 has an in-depth explanation of the Geolocation API at *http://diveintohtml5.info/geolocation.html*, whereas the MozDev article "Orientation and Motion Data Explained" gives a good overview of three-dimensional orientation and movement: *https://developer.mozilla.org/en-US/docs/DOM/Orientation_and_motion_data_explained/*.

The Fullscreen API is explained in the Sitepoint article "How to Use the HTML5 Full-Screen API" by Craig Buckler, although the API changed slightly as I was writing this, so some object names or properties may have been updated. You can find the article at *http://www.sitepoint.com/html5-full-screen-api/*.

The Battery Status API is well explained by David Walsh at *http://davidwalsh.name/battery-api/*, and a discussion of the previous and newly updated Network Information API is at *http://nostrongbeliefs.com/a-quick-look-network-information-api/*.

HTML5 Rocks gives the best explanation of `getUserMedia()` in their article "Capturing Audio & Video in HTML5": *http://www.html5rocks.com/en/tutorials/getusermedia/intro/*. The full aims of the WebRTC project are listed at *http://www.webrtc.org/*.

MozDev (again) gives a concise introduction to the Web Storage API: *https://developer.mozilla.org/en-US/docs/DOM/Storage/*.

The most accessible guide to the Drag and Drop API that I found was written by the HTML5 Doctors at *http://html5doctor.com/native-drag-and-drop/*, while the five-part "Working with Files in JavaScript" by Nicholas Zakas is an excellent resource for the File API: *http://www.nczonline.net/blog/2012/05/08/working-with-files-in-javascript-part-1/*.

The APIs that form the Firefox OS project are listed at *https://wiki.mozilla.org/WebAPI/*, and the slides from the presentation "WebAPIs and Apps" by Robert Nyman provide a great overview of the APIs: *http://www.slideshare.net/robnyman/web-apis-apps-mozilla-london/*. "Are We Mobile Yet?" gives an at-a-glance guide to levels of API implementation: *http://arewemobileyet.com/*.