HTML5 fundamentals, Part 3: The power of HTML5 APIs

Managing communication

Grace Walker  
June 07, 2011

HTML5 reflects the monumental changes in the way you now do business on the web and in the cloud. This article is the third in a four-part series designed to spotlight changes in HTML5, beginning with the new tags and page organization and providing high-level information on web page design, the creation of forms, the use and value of the APIs, and, finally, the creative possibilities that Canvas provides. This installment introduces HTML5 APIs, using an example page to demonstrate functions.

View more content in this series

So, what is an API?

An application programming interface is a collection of programming instructions and standards for accessing a software application. With an API, you can design products powered by the service the API provides.

HTML5 has several new APIs. For example:

- A 2D drawing API used with the new canvas element for rendering graphs or other visual images
- An API caching mechanism that supports offline web applications
- An API for playing video and audio used with the new video and audio elements
- A history API that makes the browsing history accessible and allows pages to add to it
- A drag-and-drop API to use with the draggable attribute
- An editing API to use with the contenteditable attribute
- Client-side storage with JavaScript APIs for key-value pairs and also embedded SQL databases

This article concentrates on two APIs: Geolocation and Web Worker. First, it analyzes the APIs themselves; then, you create a page that contains both the APIs.
Business is everywhere: Geolocation

You use the Geolocation API to determine and share geographical positions. The API returns longitude and latitude coordinates—information that businesses can use to offer services in the area approximate to the coordinates. These services are generally referred to as location-based services (LBS).

LBS refers to the geographical data sources used to identify the physical location of the instrument being monitored and, thereby, the human associated with that location. This function gives interested parties the opportunity to interact with that individual based on the market for some geolocation-centric point of interest.

Business is really about the creation of quality, utility, and value for customers, while at the same time creating economic and financial benefits for stakeholders, creditors, stockholders, employees, and vendors. The Geolocation-powered LBS makes it quite easy to track or monitor a package or a person using a non-browser device or a browser. Commercially, geolocation is all about the use of geographical assets to determine where someone or something is located, and then selling that specific set of information to anyone who wants to use it for social, commercial, or other purposes, provided there is legal permission from the owner of the information to do so.

How geolocation works

The Geolocation API is based on a new property of the global navigator object: navigator.geolocation. The JavaScript navigator object provides useful information about the visitor's browser and system. Geolocation can determine latitude and longitude using IP addresses, web-based databases, wireless network connections, and triangulation or GPS technology. It should be noted that the accuracy of Geolocation-provided information varies based on the means of obtaining the information. On occasion, and in some locations, you may not be able to get a clear geolocation reading or any data at all.

Scripts can employ the navigator.geolocation object to determine location information related to the user's hosting device. After the location information is retrieved, a position object is created and populated with the data.

The navigator.geolocation object has three methods:

- getCurrentPosition()
- watchPosition()
- clearWatch()

The getCurrentPosition() method

The getCurrentPosition() method retrieves the user's current location, but only once. When it is called by a script, the method asynchronously attempts to obtain the current location of the hosting device. Asynchronous communication means that the sender and receiver are not concurrently engaged in communication. Using asynchronous communication lets the browser continue with other activities so that it doesn't have to wait for a response from the receiving entity.

The getCurrentPosition() method can have up to three arguments:
• **geolocationSuccess.** The callback with the current position (required)
• **geolocationError.** The callback if there was an error (optional)
• **geolocationOptions.** The geolocation options (optional)

The `navigator.geolocation.getCurrentPosition()` method returns the host device's current position to the `geolocationSuccess` callback with a `Position` object as the parameter. If there is an error, the `geolocationError` callback is invoked with a `PositionError` object. You can set three properties for `geolocationOptions`: `enableHighAccuracy`, `timeout`, and `maximumAge`. These optional properties enable high accuracy if the device supports it, a timeout period by which a position should have been returned, and a maximum amount of time that a cached location can be used, respectively.

The `getCurrentPosition()` method is called as shown here:

```javascript
void navigator.geolocation.getCurrentPosition(
    geolocationSuccess, geolocationError, geolocationOptions);
```

**The watchPosition() method**

The `watchPosition()` method polls the user location on a regular basis, watching to see whether the user location has changed. It can have up to three arguments.

When `watchPosition` is called, it asynchronously starts a watch process involving the acquisition of a new `Position` object and creation of a `watchID`. If this acquisition is successful, the associated `geolocationSuccess` with a `Position` object as an argument is invoked. Upon a failure involving an invoked method with a non-null `geolocationError` argument, the method generates the `geolocationError` with a `PositionError` object as an argument. When the device position changes, a suitable callback with new `Position` object is invoked.

The `watchPosition()` method is called as shown here:

```javascript
long navigator.geolocation.watchPosition(
    geolocationSuccess, geolocationError, geolocationOptions);
```

**The clearWatch() method**

The `clearWatch()` method terminates an ongoing `watchPosition()`. This method can have only one argument. When called, it finds the `watchID` argument that was previously started and immediately stops it.

The `clearWatch()` method is called as shown here:

```javascript
void navigator.geolocation.clearWatch(watchID)
```

**Geolocation data: The Position object**

The Geolocation API returns a geographical `Position` object. This object has two properties: `timestamp` and `coords`. The `timestamp` property indicates the time of the geolocation data's creation. The `coords` property has seven attributes:
• `coords.latitude`. The estimated latitude
• `coords.longitude`. The estimated longitude
• `coords.altitude`. The estimated altitude
• `coords.accuracy`. The accuracy of the provided latitude and longitude estimates in meters
• `coords.altitudeAccuracy`. The accuracy of the provided altitude estimate in meters
• `coords.heading`. The current direction of movement for the hosting device in degrees, counting clockwise relative to true north
• `coords.speed`. The device's current ground speed in meters per second

Only three of the properties are guaranteed to be there: `coords.latitude`, `coords.longitude`, and `coords.accuracy`. The rest return `null`, depending on the capabilities of your device and the back-end positioning server that it talks to. The `heading` and `speed` properties are calculated based on the user's previous position, if possible.

**Web workers to the rescue**

Web workers remedy the problems caused by concurrency. Web workers are the HTML5 family's answer to the JavaScript single-thread problem: They run processes on a separate thread from the main page, preserving the page for the main functions, such as maintaining a stable UI.

A web worker is a JavaScript file that is loaded and executed in the background. These workers allow you to load a JavaScript file dynamically, and then execute a script using a background process that does not affect the UI. Web workers have limited access and are only allowed to pass strings. Because web workers don't use the browser UI thread, they are not permitted access to the DOM. Workers can use both `self` and `this` references for the worker's global scope. Worker and parent page communication is achieved using an event model and the `postMessage()` method.

Because web workers have a multithreaded behavior, they can only access a subset of JavaScript's features. Web workers can:

- Access the navigator object
- Use the read-only location object
- Execute `XMLHttpRequest` to send HTTP or HTTPS requests
- Set a time or interval for an activity using `setTimeout()`/`clearTimeout()` and `setInterval()`/`clearInterval()`
- Access the application cache
- Import external scripts using the `importScripts()` method
- Spawn other web workers (The child (subworker) must have the same origin as the main page and be placed in the same location as the parent worker.)

There are two types of web workers: dedicated workers and shared workers.

**Dedicated web worker**

A dedicated worker is linked to the script that created it, and it can communicate with other workers or browser components. However, it cannot communicate with the DOM.
A dedicated worker is created by passing a JavaScript file name to a new worker instance. You create a new worker using the `Worker()` constructor by specifying the worker's executing script URI. To create a dedicated worker, enter the code shown here, which creates a new dedicated `Worker` object:

```javascript
var worker = new Worker('worker.js');
```

**Shared web workers**

Shared web workers, like dedicated workers, cannot access the DOM and have only limited access to window properties. Shared web workers can only communicate with other shared web workers from the same domain. The workers are created by passing a JavaScript name to a new shared worker instance.

Page scripts can communicate with shared web workers. However, unlike dedicated web workers, you communicate by using a `port` object and attaching a message event handler. In addition, you must call the port's `start()` method before using the first `postMessage()`.

Upon receipt of the first message by the web worker script, the shared web worker attaches an event handler to the active port. Generally, the handler will run its own `postMessage()` method to return a message to the calling code, and then the port's `start()` method generates an enable message process.

To create a shared web worker, you must create a `SharedWorker` object instead of the `Worker` object. The following code shows how a new `SharedWorker` object is created:

```javascript
var worker = new SharedWorker('worker.js');
```

**Constructing a page including the two APIs**

You will design a page that contains basic working models of the Geolocation and Web Worker APIs. In addition, you use the Google Map API to render the data gathered as a map.

The page is organized as shown in Figure 1. It contains a Header area created using the `<header></header>` tags, a Section area created using the `<section></section>` tags, and an Aside area created using the `<aside></aside>` tags.

**Figure 1. API page layout**

The `<section>` and `<aside>` areas contain the APIs. The Section area contains the Geolocation API. The Aside area contains the web worker, which calculates prime numbers.
When executed, the web page is displayed as shown in Figure 2. To view the geolocation data, you must first agree to share your information. The web worker starts when the page loads. If you want to see the prime numbers found, click Display Web Worker.

**Figure 2. The API web page**

The HTML file

The HTML file begins with the standard HTML5 information shown in Listing 1. The `<head>` section contains a call to the Google Maps API, setting the value of sensor to False. Using the Google Maps API requires that you state whether your application is using a sensor, such as a GPS, to establish location. You must declare a sensor parameter value of True or False for your Google Maps API application. A sensor value must be declared. The `<head>` tag also contains links to the JavaScript and CSS3 files used to handle the functions and format the web page.

**Listing 1. HTML file beginning**

```html
<!doctype html>
<html>
<head>
<title>Basic GeoLocation Map & Web Worker Prime Number Calculator</title>
<script src="http://maps.google.com/maps/api/js?sensor=false" type="text/javascript"></script>
<link href="GeolocationWebWorker.css" rel="stylesheet" type="text/css">
<script src="HTML-Part3-GeolocationWebWorker.js" type="text/javascript"></script>
</head>
```

The `<body>` tag contains an `onLoad` event that calls the initialization function for geolocation, as shown in Listing 2. This function verifies that geolocation can be used in this browser. The initialization function is in the JavaScript file. If the browser can communicate with the Geolocation API, the map will be rendered.
Listing 2. Initialize Geolocation

```html
<body onLoad="initGeoApp();">
<header>
  <hgroup>
    <h1>Geolocation & Web Worker</h1>
    <h2>Making it work</h2>
  </hgroup>
</header>
```

The `<section>` tag shown in Listing 3 contains the display output information for the `navigator.geolocation` object. A map canvas is created using the longitude and latitude that the API returns. The `Position coords` data is also displayed using the `<span>` tags.

Listing 3. Geolocation map and position

```html
<section>
  <p>This is the geolocation example map.</p>
  <div id="map_canvas"></div>
  <p>This is the output from the navigator.geolocation object.</p>
  <table>
    <tr>
      <td>accuracy:</td>
      <td><span id="accuracyOutput"></span></td>
    </tr>
    <tr>
      <td>altitude:</td>
      <td><span id="altitudeOutput"></span></td>
    </tr>
    <tr>
      <td>altitudeAccuracy:</td>
      <td><span id="altitudeAccuracyOutput"></span></td>
    </tr>
    <tr>
      <td>heading:</td>
      <td><span id="headingOutput"></span></td>
    </tr>
    <tr>
      <td>latitude:</td>
      <td><span id="latitudeOutput"></span></td>
    </tr>
    <tr>
      <td>longitude:</td>
      <td><span id="longitudeOutput"></span></td>
    </tr>
    <tr>
      <td>speed:</td>
      <td><span id="speedOutput"></span></td>
    </tr>
  </table>
</section>
```

The Web Worker calculates prime numbers. You use the new `<output>` tag to display the calculation that the web worker provides. The ID assigned in the `<output>` tag is the same ID JavaScript uses to identify the calculation it performs. The IDs used in the `<span>` and `<output>` tags makes them accessible to the DOM. Without the reference ID, JavaScript will not know which `<span>` or `<output>` to use. Listing 4 shows the output from the web worker.
Listing 4. Web worker output

```
<aside>
  <p>This is the Web Worker. </p>
  <p>Prime number calculation result:
      <output id="result"></output></p>
</aside>
```

The `onClick` is used in the `<input>` tag to first display the values being calculated by the Prime Number web worker, and then the second `onClick` is used to stop the web worker. Listing 5 shows the code. The `displayWorker()` function causes the web worker's calculations to be displayed when the button is clicked. The web worker began calculating the prime numbers when the page was loaded.

Listing 5. Inputs for the web worker

```
<input type="button" value="Display Web Worker" onClick="displayWorker();">
<input type="button" value="Stop Web Worker" onClick="stopWorker();">
```

The JavaScript file

JavaScript is the engine behind the APIs exhibited on the example page. The Geolocation API is initialized with the `initGeoApp()` function. This is the function executed by the `onLoad()` event in the `<body>` tag: It determines whether your browser can use geolocation (see Listing 6). If your browser can use geolocation, then the Geolocation API is called. If successful, a map is drawn using the `Position` attributes. The values of the attributes are then printed below the map.

Listing 6. Geolocation functions

```
function initGeoApp()
{
  if( navigator.geolocation )
  {
    navigator.geolocation.getCurrentPosition( success, failure);
  }
else
  {
    alert("Your browser does not support geolocation services.");
  }
}
```

The values are retrieved using `document.getElementById`, based on the ID that you supplied in the HTML file. `document.getElementById` is a method of the document object and should be accessed by using `document.getElementById`, as shown in Listing 7. The values of the `Position` attributes are stored here so that they can be used to print the attributes below the map to be rendered.
Listing 7. Use `getElementById` to get coords values

```javascript
var map;
function success(position)
{
    document.getElementById("accuracyOutput").innerHTML = position.coords.accuracy;
    document.getElementById("altitudeOutput").innerHTML = position.coords.altitude;
    document.getElementById("altitudeAccuracyOutput").innerHTML = position.coords.altitudeAccuracy;
    document.getElementById("headingOutput").innerHTML = position.coords.heading;
    document.getElementById("latitudeOutput").innerHTML = position.coords.latitude;
    document.getElementById("longitudeOutput").innerHTML = position.coords.longitude;
    document.getElementById("speedOutput").innerHTML = position.coords.speed;
}
```

This section defines the coordinates for the Google Map API's `LatLng` object, as Listing 8 shows. The Google Map API `LatLng` object provides the coordinate information required to create a map. You can set the zoom level and several other options that create the look of the map presented to the user.

Listing 8. Google Map options

```javascript
var coordinates = new google.maps.LatLng(position.coords.latitude, position.coords.longitude);

var myOptions =
{
    zoom: 14,
    center: coordinates,
    mapTypeControl: false,
    navigationControlOptions: {style: google.maps.NavigationControlStyle.small},
    mapTypeId: google.maps.MapTypeId.ROADMAP
};
```

Note that in the `mapTypeId` option, the option selected is the ROADMAP. This value presents the map so that it appears as shown in Figure 2. There are four possible values:

- ROADMAP
- HYBRID
- SATELLITE
- TERRAIN

Figure 3 shows how the page would appear with the **HYBRID** option selected.
Create the map using the ID map_canvas, which is the ID for the <div> in the HTML file:

```javascript
map = new google.maps.Map(document.getElementById("map_canvas"), myOptions);
```

Place an initial position marker on the map. Listing 9 shows the code.

Listing 9. Place an initial map marker

```javascript
var marker = new google.maps.Marker({
  position: coordinates,
  map: map,
  title: "You are here."
});

function failure()
{
  alert("Sorry, could not obtain location");
}
```

The web worker begins executing when the page is initialized. If the user wants to display the output of the calculations being performed, he or she can click Display Web Worker, which will call the displayWorker() function. Listing 10 shows the code.

Listing 10. The web worker

```javascript
var worker = new Worker('PrimeNumberWebWorker.js');

function displayWorker()
{
  worker.onmessage = function (event)
  {
    document.getElementById('result').innerHTML = event.data;
  }
}```
If the user wants to stop the web worker, he or she can click **Stop Web Worker**, which will call the `stopWorker()` function shown in Listing 11.

### Listing 11. Terminate worker

```javascript
function stopWorker()
{
    worker.terminate();
}
```

### The web worker file

This file is the prime number calculator web worker: It calculates every prime number until it is stopped. Listing 12 shows the code.

### Listing 12. Calculate prime numbers

```javascript
var n = 1;
search: while (true) {
    n += 1;
    for (var i = 2; i <= Math.sqrt(n); i += 1)
        if (n % i == 0)
            continue search;
    postMessage(n);
}
```

### The CSS3 file

The CSS3 file shown in Listing 13 provides the formatting displayed in the HTML5 page.

### Listing 13. CSS3 descriptions

```css
* {font-family: Arial,Helvetica,sans-serif ;
}
body {
    margin: 0 300px 0 300px;
    color: #990000;
    background-color:#FFFFCC;
}
header > hgroup h1 {
    margin: 0 0 3px 0;
    padding: 0;
    text-align: center;
    font-size: 30px;
}
header > hgroup h2 {
    margin: 0 0 15px 0;
    padding: 0;
    text-align: center;
    font-style: italic;
    font-size: 12px;
}
```
Conclusion

This installment examined the utility of the Geolocation and Web Worker APIs. These two APIs were selected because together they demonstrate both the innovative and the practical use of APIs. Geolocation is an excellent example of the HTML5 specification's use in the creation of new business models. Likewise, the Web Worker's role is the resolution of the problems inherent in JavaScript's concurrency problem.

These two APIs together illustrate a model combination of the use of HTML5 for commercial and social use. Thus, their utility demonstrates the proper facilitation and general management of an HTML5 rich Internet application.
### Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example HTML, CSS3 and JavaScript files</td>
<td>HTML5APIs.zip</td>
<td>10KB</td>
</tr>
</tbody>
</table>

© Copyright IBM Corporation 2011  
**Trademarks**  
(www.ibm.com/developerworks/ibm/trademarks/)