Build Web applications with HTML 5

Create tomorrow's Web applications today

Skill Level: Intermediate

Michael Galpin (mike.sr@gmail.com)
Software architect
eBay

30 Mar 2010

For years Web developers have salivated over some of the features promised in the next generation of Web browsers as outlined by the HTML 5 specification. You might be surprised to learn just how many of the features are already available in today's browsers. In this article, learn how to detect which capabilities are present and how to take advantage of those features in your application. Explore powerful HTML 5 features such as multi-threading, geolocation, embedded databases, and embedded video.

Introduction

Many new features and standards have emerged as part of HTML 5. Once you detect the available features in today's browsers, you can take advantage of those features in your application. In this article, learn how to detect and use the latest Web technologies by developing sample applications. Most of the code in this article is just HTML, JavaScript, and CSS—the core technologies of any Web developer.

Getting started

To follow along with the examples, the most important thing you need is multiple browsers for testing. The latest versions of Mozilla Firefox, Apple Safari, and Google Chrome are highly recommended. Mozilla Firefox 3.6, Apple Safari 4.04, and Google Chrome 5.0.322 were used for this article. You might also want to test on mobile browsers. For example, the latest Android and iPhone SDKs were used for testing their browsers on their emulators.

You can download the source code used in this article.

The examples include a very small back-end component that was written in Java™. JDK 1.6.0_17 and Apache Tomcat 6.0.14 were used. See Resources for links to download the tools.

Detecting capabilities

There's an old joke that Web developers spend about 20% of their time writing code and the other 80% getting it to work the same in all browsers. To say that Web developers are used to dealing with cross-browser differences is an understatement. With a new wave of browser innovations unfolding, this pessimistic approach is once again warranted. The features supported by the latest and greatest browsers are always changing.

On a positive note, however, the new feature sets are converging on Web standards, which gives you a chance to start using these new features today. You can: employ the old technique of progressive enhancement, provide some baseline features, check for advanced capabilities, and then enhance your applications with the extra features when they are present. To that end, take a look at how to detect some of the new features. Listing 1 shows a simple detection script.

Listing 1. Detection script

```
function detectBrowserCapabilities(){
    $("userAgent").innerHTML = navigator.userAgent;
    var hasWebWorkers = !!window.Worker;
    $("workersFlag").innerHTML = "" + hasWebWorkers;
    var hasGeolocation = !!navigator.geolocation;
    $("geoFlag").innerHTML = "" + hasGeolocation;
    if (hasGeolocation){
        document.styleSheets[0].cssRules[1].style.display = "block";
        navigator.geolocation.getCurrentPosition(function(location) {
            $("geoLat").innerHTML = location.coords.latitude;
            $("geoLong").innerHTML = location.coords.longitude;
        });
    }
    var hasDb = !!window.openDatabase;
    $("dbFlag").innerHTML = "" + hasDb;
    var videoElement = document.createElement("video");
    var hasVideo = !!videoElement["canPlayType"];
    var ogg = false;
    var h264 = false;
    if (hasVideo) {
        ogg = videoElement.canPlayType('video/ogg; codecs="theora, vorbis"') || "no";
           h264 = videoElement.canPlayType('video/mp4;
codecs="avc1.42E01E, mp4a.40.2"') || "no";
    }
    $("videoFlag").innerHTML = "" + hasVideo;
    if (hasVideo){
        var vStyle = document.styleSheets[0].cssRules[0].style;
        vStyle.display = "block";
    }
    $("h264Flag").innerHTML = "" + h264;
    $("oggFlag").innerHTML = "" + ogg;
```

```
}
```

A huge number of new features and standards have emerged as part of the HTML 5 standard. This article focuses on only a few of the rather useful features. The script in Listing 1 detects four features:

- Web workers (multi-threading)
- Geolocation
- Database storage
- Native video playback

The script starts by showing the user agent of the user's browser. This is (usually) a string that uniquely identifies the browser, though it can be easily faked. Just echoing it is good enough for this application. The next step is to start detecting features. First, check for Web workers by looking for the `Worker` function in the global scope (window). This is using some idiomatic JavaScript: the double negation. If the `Worker` function does not exist, then `window.Worker` will evaluate to undefined, which is a "falsey" value in JavaScript. Putting a single negation in front of it will evaluate to true, thus a double negation will evaluate to false. After testing for the value, the script prints the evaluation to the screen by modifying the DOM structure shown in Listing 2.

Listing 2. Detection DOM

```
<input type="button" value="Begin detection"
    onclick="detectBrowserCapabilities()"/>
<div>Your browser's user-agent: <span id="userAgent">
   </span></div>
<div>Web Workers? <span id="workersFlag"></span></div>

<div>Database? <span id="dbFlag"></span></div>
<div>Video? <span id="videoFlag"></span></div>
<div class="videoTypes">Can play H.264? <span id="h264Flag">
   </span></div>
<div class="videoTypes">Can play OGG? <span id="oggFlag">
   </span></div>
<div>Geolocation? <span id="geoFlag"></span></div>
<div class="location">
    <div>Latitude: <span id="geoLat"></span></div>
    <div>Longitude: <span id="geoLong"></span></div>
</div>
```

Listing 2 is a simple HTML structure used to display the diagnostic information that the detection script is gathering. As shown in Listing 1, the next thing to test for is geolocation. The double negation technique is used again, but this time you're checking for an object called `geolocation` that should be a property of the `navigator` object. If it is present, then use it to get the current location using the `geolocation` object's `getCurrentPosition` function. Getting the location can

be a slow task, since it usually involves scanning Wi-Fi networks. On mobile devices, it might also include scanning cell towers and pinging GPS satellites. Since it could take a long time, the `getCurrentPosition` is asynchronous and takes a `callback` function as a parameter. In this case, use a closure for the `callback` that simply displays the location fields (by toggling its CSS) and then writes the latitude and longitude to the DOM.

The next step is to check for database storage. Check for the presence of the global function `openDatabase`, which is used for creating and accessing client-side databases.

Finally, check for native video playback. Use the DOM API to create a video element. Today, every browser will be able to create such an element. In older browsers this will be a valid DOM element, but it will have no special meaning. It would be like creating an element called `foo`. In a modern browser, this will be a specialized element, like creating a `div` or `table` element. It will have a function called `canPlayType`, so simply check its presence.

Even if a browser has native video playback capability, the types of videos, or the supported codecs that it can playback, are not standardized. You'll probably want to check for the supported codecs in the browser. There is no standard list of codecs, but two of the most common are H.264 and Ogg Vorbis. To check for support for a particular codec, you can pass an identifying string to the `canPlayType` function. If the browser can support the codec, this function will return `probably` (seriously—that's not a joke). If not, then it will return null. In the detection code, simply check against these values and display the answer in the DOM. After testing out this code against some popular browsers, Listing 3 shows aggregated results.

Listing 3. Capabilities of various browsers

```
#Firefox 3.6
Your browser's user-agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.6;
en-US; rv:1.9.2) Gecko/20100115 Firefox/3.6
Web Workers? true
Database? false
Video? true
Can play H.264? no
Can play OGG? probably
Geolocation? true
Latitude: 37.2502812
Longitude: -121.9059866

#Safari 4.0.4
Your browser's user-agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_2;
en-us) AppleWebKit/531.21.8 (KHTML, like Gecko) Version/4.0.4 Safari/531.21.10
Web Workers? true
Database? true
Video? true
Can play H.264? probably
Can play OGG? no
Geolocation? false

#Chrome 5.0.322
```

```
Your browser's user-agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_2;
en-US) AppleWebKit/533.1 (KHTML, like Gecko) Chrome/5.0.322.2 Safari/533.1
Web Workers? true
Database? true
Video? true
Can play H.264? no
Can play OGG? no
Geolocation? false
```

All of the popular desktop browsers above support quite a few features.

- Firefox supports everything except for databases. For video, it only supports Ogg.
- Safari supports everything except for geolocation.
- Chrome supports everything except for geolocation, though it claims not to support H.264 or Ogg. This is probably a bug, either with the build of Chrome used for this test or with the test code. Chrome actually does support H.264.

Geolocation is not widely supported on desktop browsers, but it is widely supported on mobile browsers. Listing 4 shows aggregated results for mobile browsers.

Listing 4. Mobile browsers

```
#iPhone 3.1.3 Simulator
Your browser's user-agent: Mozilla/5.0 (iPhone Simulator; U; CPU iPhone OS 3.1.3
like Mac OS X; en-us) AppleWebKit/528.18 (KHTML, like Gecko)
Version/4.0 Mobile/7E18 Safari/528.16
Web Workers? false
Database? true
Video? true
Can play H.264? maybe
Can play OGG? no
Geolocation? true
Latitude: 37.331689
Longitude: -122.030731

#Android 1.6 Emulator
Your browser's user-agent: Mozilla/5.0 (Linux; Android 1.6; en-us;
sdk Build/Donut) AppleWebKit/528.5+ (KHTML, like Gecko) Version/3.1.2
Mobile Safari/525.20.1
Web Workers? false
Database? false
Video? false
Geolocation? false

#Android 2.1 Emulator
Your browser's user-agent: Mozilla/5.0 (Linux; U; Android 2.1; en-us;
sdk Build/ERD79) AppleWebKit/530.17 (KHTML, like Gecko) Version/4.0
Mobile Safari/530.17
Web Workers? true
Database? true
Video? true
Can play H.264? no
Can play OGG? no
Geolocation? true
Latitude:
```

```
Longitude:
```

One of the latest iPhone simulators and two flavors of Android are shown above. Android 1.6 does not support anything that we tested for. It does, in fact, support all of these features except for video but it does so using Google Gears. These are equivalent APIs (in terms of function), but they do not conform to Web standards so you get the result in Listing 4. Compare this with Android 2.1, where everything is supported.

Notice that the iPhone supports everything but Web workers. Listing 3 shows that the desktop version of Safari supports Web workers, so it would seem reasonable to expect that this feature will be coming to the iPhone very soon.

Now that you've seen how to probe the features of the user's browser, let's explore a simple application that will use several of these features in combination—depending on what the user's browser can handle. You will build an application that uses the Foursquare API to search for popular venues near a user's location.

Building the applications of tomorrow

The example will focus on using geolocation on mobile devices, but keep in mind that Firefox 3.5+ also supports geolocation. The application starts off by searching for what Foursquare calls *venues* near the user's current location. Venues can be anything, but are typically restaurants, bars, stores, and so on. Being a Web application, our example is limited by the same origin policy enforced by all browsers. It cannot call Foursquare's APIs directly. You will use a Java servlet to essentially proxy these calls. There is nothing special about Java here; you could easily write a similar proxy in PHP, Python, Ruby, and so forth. Listing 5 shows a proxy servlet.

Listing 5. Foursquare proxy servlet

```
public class FutureWebServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {
        String operation = request.getParameter("operation");
        if (operation != null && operation.equalsIgnoreCase("getDetails")){

            getDetails(request,response);
        }
        String geoLat = request.getParameter("geoLat");
        String geoLong = request.getParameter("geoLong");
        String baseUrl = "http://api.foursquare.com/v1/venues.json?";
        String urlStr = baseUrl + "geolat=" + geoLat + "&geolong=" + geoLong;
        PrintWriter out = response.getWriter();
        proxyRequest(urlStr, out);
    }

    private void proxyRequest(String urlStr, PrintWriter out) throws IOException{
        try {
```

```
            URL url = new URL(urlStr);
            InputStream stream = url.openStream();
            BufferedReader reader = new BufferedReader( new InputStreamReader(stream));
            String line = "";
            while (line != null){
                line = reader.readLine();
                if (line != null){
                    out.append(line);
                }
            }
            out.flush();
            stream.close();
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
    }

    private void getDetails(HttpServletRequest request, HttpServletResponse response)
throws IOException{
        String venueId = request.getParameter("venueId");
        String urlStr = "http://api.foursquare.com/v1/venue.json?vid="+venueId;
        proxyRequest(urlStr, response.getWriter());
    }
}
```

The important thing to note here is that you're proxying two Foursquare APIs. One is for search, and the other is for getting the details of a venue. To distinguish between them, the details API adds an operation parameter. You're also specifying JSON as the return type, which will make it easy to parse the data from JavaScript. Now that you know what kind of calls can be made by the application code, let's see how it will make those calls and use the data from Foursquare.

Using geolocation

The first call is a search. Listing 5 shows that you need two parameters: `geoLat` and `geoLong` for the latitude and longitude. Listing 6 below shows how to get these in the application and call the servlet.

Listing 6. Calling search with location

```
if (!!navigator.geolocation){
    navigator.geolocation.getCurrentPosition(function(location) {
        venueSearch(location.coords.latitude, location.coords.longitude);
    });
}
var allVenues = [];
function venueSearch(geoLat, geoLong){
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function(){
        if (this.readyState == 4 && this.status == 200){
            var responseObj = eval('(' + this.responseText + ')');
            var venues = responseObj.groups[0].venues;
            allVenues = venues;
            buildVenuesTable(venues);
        }
    }
    xhr.open("GET", "api?geoLat=" + geoLat + "&geoLong="+geoLong);
    xhr.send(null);
}
```

The code above checks for the geolocation capability of the browser. If it is present, the code gets the location and calls the `venueSearch` function with the latitude and longitude. This function uses Ajax (an `XMLHttpRequest` object to call the servlet in Listing 5). It uses a closure for the `callback` function, parses the JSON data from Foursquare, and passes an array of venue objects to a function called `buildVenuesTable`, as shown below.

Listing 7. Building UI from venues

```
function buildVenuesTable(venues){
    var rows = venues.map(function (venue) {
        var row = document.createElement("tr");
        var nameTd = document.createElement("td");
        nameTd.appendChild(document.createTextNode(venue.name));
        row.appendChild(nameTd);
        var addrTd = document.createElement("td");
        var addrStr = venue.address + " " + venue.city + "," + venue.state;
        addrTd.appendChild(document.createTextNode(addrStr));
        row.appendChild(addrTd);
        var distTd = document.createElement("td");
        distTd.appendChild(document.createTextNode("" + venue.distance));
        row.appendChild(distTd);
        return row;
    });
    var vTable = document.createElement("table");
    vTable.border = 1;
    var header = document.createElement("thead");
    var nameLabel = document.createElement("td");
    nameLabel.appendChild(document.createTextNode("Venue Name"));
    header.appendChild(nameLabel);
    var addrLabel = document.createElement("td");
    addrLabel.appendChild(document.createTextNode("Address"));
    header.appendChild(addrLabel);
    var distLabel = document.createElement("td");
    distLabel.appendChild(document.createTextNode("Distance (m)"));
    header.appendChild(distLabel);
    vTable.appendChild(header);
    var body = document.createElement("tbody");
    rows.forEach(function(row) {
        body.appendChild(row);
    });
    vTable.appendChild(body);
    $("searchResults").appendChild(vTable);
    if (!!window.openDatabase){
        $("saveBtn").style.display = "block";
    }
}
```

The code in Listing 7 is primarily DOM code for creating a data table with the venue information in it. There are a few interesting things, though. Note the use of advanced JavaScript features, such as the array object's map and `forEach` functions. These are features that are available on all the browsers that support geolocation. Also of interest are the last two lines. A detection for database support is performed. If it is present, then you enable a Save button that the user can click to save all of this venue data to a local database. The next section discusses how this is done.

Structured storage

Listing 7 demonstrates the classic progressive enhancement strategy. The example tests for database support. If it is found, then the code adds a UI element that adds a new feature to the application that makes use of this feature. In this case, it enables a single button. Clicking on the button calls the function `saveAll`, which is shown in Listing 8.

Listing 8. Saving to the database

```
var db = {};
function saveAll(){
    db = window.openDatabase("venueDb", "1.0", "Venue Database",1000000);
    db.transaction(function(txn){
        txn.executeSql("CREATE TABLE venue (id INTEGER NOT NULL PRIMARY KEY, "+
                "name NVARCHAR(200) NOT NULL, address NVARCHAR(100),
cross_street NVARCHAR(100), "+
                "city NVARCHAR(100), state NVARCHAR(20), geolat TEXT NOT NULL, "+
                "geolong TEXT NOT NULL);");
    });
    allVenues.forEach(saveVenue);
    countVenues();
}
function saveVenue(venue){
    // check if we already have the venue
    db.transaction(function(txn){
        txn.executeSql("SELECT name FROM venue WHERE id = ?", [venue.id],
            function(t, results){
                if (results.rows.length == 1 && results.rows.item(0)['name']){
                    console.log("Already have venue id=" + venue.id);
                } else {
                    insertVenue(venue);
                }
            })
    });
}
function insertVenue(venue){
    db.transaction(function(txn){
        txn.executeSql("INSERT INTO venue (id, name, address, cross_street, "+
                "city, state, geolat, geolong) VALUES (?, ?, ?, ?, "+
                "?, ?, ?, ?);", [venue.id, venue.name,
                 venue.address, venue.crossstreet, venue.city, venue.state,
                 venue.geolat, venue.geolong], null, errHandler);
    });
}
function countVenues(){
    db.transaction(function(txn){
        txn.executeSql("SELECT COUNT(*) FROM venue;",[], function(transaction, results){
            var numRows = results.rows.length;
            var row = results.rows.item(0);
            var cnt = row["COUNT(*)"];
            alert(cnt + " venues saved locally");
        }, errHandler);
    });
}
```

To save the venue data to the database, start by creating the table where you want to store the data. This is fairly standard SQL syntax for creating a table. (All the browsers that support databases use SQLite. Use SQLite documentation for data types supported, constraints, and so on.) SQL execution is done asynchronously.

The transaction function is called, and a callback function is passed to it. The `callback` function gets a transaction object that it can use to execute SQL. The `executeSQL` function takes an SQL string, then optionally a parameter list, plus success and error handler functions. If there is no error handler, the error is "eaten." For the `create table` statement this is desirable. The first time the script executes, the table will be properly created. The next time it executes, the script will fail since the table already exists—but that's okay. You just need to make sure the table is there before you start inserting rows into it.

After the table is created, use the `forEach` function to invoke the `saveVenue` function with each of the venues returned from Foursquare. This function first checks to see if the venue has already been stored locally by querying for it. Here you see the use of a success handler. The result set from the query will be passed to the handler. If there are no results, or the venue has not already been saved locally, then call the `insertVenue` function, which performs an insert statement.

With `saveAll`, after all of the save/inserts are complete, you then call `countVenues`. This queries to see the total number of rows that have been inserted into the venue table. The syntax here (`row["COUNT(*)"]`) pulls out the count from the result set of the query.

Now that you've learned how to use database support if it is present, the next section explores how to use Web worker support.

Background processing with Web workers

Going back to Listing 6, let's make a slight modification. As shown in Listing 9 below, check for Web worker support. If it is there, use it to get more information on each of the venues retrieved from Foursquare.

Listing 9. Modified venue search

```
function venueSearch(geoLat, geoLong){
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function(){
        if (this.readyState == 4 && this.status == 200){
            var responseObj = eval('(' + this.responseText + ')');
            var venues = responseObj.groups[0].venues;
            allVenues = venues;
            buildVenuesTable(venues);
            if (!!window.Worker){
                var worker = new Worker("details.js");
                worker.onmessage = function(message){
                    var tips = message.data;
                    displayTips(tips);
                };
                worker.postMessage(allVenues);
            }
        }
    }
    xhr.open("GET", "api?geoLat=" + geoLat + "&geoLong="+geoLong);
    xhr.send(null);
}
```

The code above uses the same detection you've seen before. If Web workers are supported, then you create a new worker. To create a new worker, you need a URL to another script that the worker will execute—the details.js file, in this case. When the worker finishes its work, it will send a message back to the main thread. The `onmessage` handler is what will receive this message; you use a simple closure for it. Finally, to initiate the worker, call `postMessage` with some data for it to work on. You're passing in all of the venues retrieved from Foursquare. Listing 10 shows the contents of details.js, which is the script that will be executed by the worker.

Listing 10. The worker's script, details.js

```
var tips = [];
onmessage = function(message){
    var venues = message.data;
    venues.foreach(function(venue){
        var xhr = new XMLHttpRequest();
        xhr.onreadystatechange = function(){
            if (this.readyState == 4 && this.status == 200){
                var venueDetails = eval('(' + this.responseText + ')');
                venueDetails.tips.forEach(function(tip){
                    tip.venueId = venue.id;
                    tips.push(tip);
                });
            }
        };
        xhr.open("GET", "api?operation=getDetails&venueId=" + venueId, true);
        xhr.send(null);
    });
    postMessage(tips);
}
```

The details script iterates over each of the venues. For each venue, the script then makes a call back to the Foursquare proxy to get the details of the venue using `XMLHttpRequest`, as usual. However, notice that when you use its `open` function to open the connection, you pass a third parameter (`true`). This makes the call synchronous instead of the usual asynchronous. It's okay to do this from a worker since you are not on the main UI thread, and this is not going to freeze up the application. By making it synchronous, you know that each call has to finish before the next one begins. The handler is simply extracting the tips from the venue details and collecting all of these tips to be passed back to the main UI thread. To pass this data back, the `postMessage` function is called, which will invoke the `onmessage` callback function on the worker, as shown back in Listing 9.

By default, the venue search returns ten venues. You can imagine how long it would take to make the ten extra calls to get the details. It makes sense to do this type of task in a background thread using Web workers.

Summary

This article covered some of the new HTML 5 capabilities of modern browsers. You learned how to detect the features and to progressively add them to your application. Most of the features are already widely supported in popular browsers—especially mobile browsers. Now you can start taking advantage of things like geolocation and Web workers to create innovative new Web applications.

Downloads

Description	Name	Size	Download method
Article source code	FutureWeb.zip	9KB	HTTP

Information about download methods

Resources

Learn

- "Explore multithreaded programming in XUL" (Michael Galpin, developerWorks, Sep 2009) has more information about using Web workers.
- "New elements in HTML 5" (developerWorks, Aug 2007) explores some of the many UI features in the HTML 5 spec.
- "Android and iPhone browser wars, Part 1: WebKit to the rescue" (developerWorks, Dec 2009) explains how you can leverage the features of mobile browsers.
- The W3C HTML 5 Specification is the definitive source on HTML 5.
- "DOM Storage" (Mozilla Developer Center) discusses HTML 5's localStorage support in Firefox.
- The developerWorks Web development zone specializes in articles covering various Web-based solutions.
- My developerWorks: Personalize your developerWorks experience.
- To listen to interesting interviews and discussions for software developers, check out developerWorks podcasts.
- developerWorks technical events and webcasts: Stay current with developerWorks technical events and webcasts.

Get products and technologies

- The Modernizr project provides a comprehensive utility for detecting HTML 5 features.
- Get Mozilla Firefox 3.6.
- Get Safari 4.04.
- Get Google Chrome 5.0.322.
- Download the Android SDK, access the API reference, and get the latest news on Android.
- Get the latest iPhone SDK. Version 3.1.3 was used in this article.
- Get the Android source code from the Android Open Source Project.
- Get the Java SDK. JDK 1.6.0_17 was used in this article.
- Innovate your next open source development project with IBM trial software, available for download or on DVD.

Discuss

- Participate in developerWorks blogs and get involved in the developerWorks community.

About the author

Michael Galpin
Michael Galpin is an architect at eBay and is a frequent contributor to developerWorks. He has spoken at various technical conferences, including JavaOne, EclipseCon and AjaxWorld. To get a preview of what he is working on, follow @michaelg on Twitter.