

jQuery Mobile

Integration with the Phone

Lesson 1, Activity 2: Integration with the Phone

Calls, Emails & Texts

Links or buttons that open the mobile device's phone, email, or SMS-text application are easy to create. The traditional `mailto:` link works fine on phones, while `tel:` and `sms://` generate calls and texts, respectively:

Call, Email & Text Links

Type	Markup
Phone Call	<code>Call 123-456-7890</code>
Email	<code>Email name@example.com</code>
Text Message	<code>SMS 123-456-7890</code>

[Apple recommends](#) the format 1-xxx-xxx-xxxx for US-based phone numbers.

In this quick example, we add "contact us" information for a page on the Nan & Bob's site, helpfully presented to users as three buttons - one each for calling, emailing, and texting:



Open [IntegrationPhone/Demos/callemailtext.html](#) to view the code and test the page on a mobile device. Note that the "call" and "text" links won't work on an iOS simulator.

Code Sample:

[IntegrationPhone/Demos/callemailtext.html](#)

```

---- C O D E   O M I T T E D ----

<a href="tel:1-123-456-7890" data-role="button">Call</a>
<a href="mailto:nanandbob@example.com" data-role="button">Email</a>
<a href="sms://1234567890" data-role="button">Send Text</a>
---- C O D E   O M I T T E D ----

```

We present each link as a button (with attribute `data-role="button"`), using `tel:` for the phone button, `mailto:` for the email button, and `sms://` for the text button.

Lesson 1, Activity 3: Custom Home Screen/Bookmark Icons

iPhone and other devices offer users the ability to add a shortcut/bookmark to a webpage on their home screen; our job as developers is to provide a custom icon that best represents our site or page. Including a line of code in the head of each page or including a specifically named icon in the website's root directory provides the functionality on iPhones and iPads; similar strategies work on Androids and other phones.

iPhone/iPad

1. Place a PNG icon, named `apple-touch-icon.png` or `apple-touch-icon-precomposed.png`, in the website root directory to specify the icon for the entire site if you don't want to add a link tag to pages; Safari on iOS won't add effects (rounded corners etc.) to the precomposed icon.
2. Instead of adding an image to the website root directory, you can instead add a link element to the head section of any page to specify an icon for a single page:

```
<link rel="apple-touch-icon" href="/custom_icon.png"/>
```

3. To support different device resolutions (iPad vs. iPhone, for instance), add a size attribute to each link element:

```
<link rel="apple-touch-icon" href="touch-icon-iphone.png" />
<link rel="apple-touch-icon" sizes="72x72" href="touch-icon-ipad.png" />
<link rel="apple-touch-icon" sizes="114x114" href="touch-icon-iphone4.png" />
```

The icon that is the most appropriate size for the device is used. If no sizes attribute is set, the element's size defaults to 57 x 57.

4. For iPhone and iPod touch, create icons that measure:

- 57 x 57 pixels
- 114 x 114 pixels (high resolution)

For iPad, create icons that measure:

- 72 x 72 pixels
- 144 x 144 (high resolution)

5. The smallest icon larger than the recommended size is used if no icon matches the recommended size. If there are no icons larger than the recommended size, the largest icon is used. If multiple icons are suitable, the icon that has the precomposed keyword is used.
6. If no icons are specified using a link element, the root directory is searched for icons with the `apple-touch-icon` or `apple-touch-icon-precomposed` prefix.

See the [Apple Developer](#) site for full details.

Other Devices

Android, Blackberry, and other devices have adopted the `apple-touch-icon` as a *de facto* standard. Please note that some versions of the Android require an absolute path, including the domain name, for the icon to work properly:

```
<link rel="apple-touch-icon" href="http://www.example.com/custom_icon.png"/>
```

instead of

```
<link rel="apple-touch-icon" href="/custom_icon.png"/>
```

Lesson 1, Activity 5: The Geolocation API

One can easily see the value of location-aware websites; increasingly, we rely on our phones to find the nearest gas station, restaurant, or shopping center. As we move through this lesson, we will explore how location-aware code might enhance the user experience for visitors to the Nan & Bob's site.

The World Wide Web Consortium (W3C) has worked to standardize a system by which developers can retrieve a user's location. Based in part on Google Gears, the API is now widely supported (see below for specifics) and offers a device-independent system for accessing the user's latitude and longitude via JavaScript.

How It Works

The GeoLocation API can find a user's location from a variety of information sources, using the best available. The W3C specification states:

The Geolocation API defines a high-level interface to location information associated only with the device hosting the implementation, such as latitude and longitude. The API itself is agnostic of the underlying location information sources. Common sources of location information include Global Positioning System (GPS) and location inferred from network signals such as IP address, RFID, WiFi and Bluetooth MAC addresses, and GSM/CDMA cell IDs, as well as user input. [From dev.w3.org/geo/api/spec-source.html#introduction]

If GPS is available, it will be used (though some devices, because of the delay and high energy-use of GPS, opt not to use GPS). If GPS is not available, A-GPS (Assistive GPS, triangulated from cellphone towers), WiFi, IP, or another less accurate means of locating the user will be used.

Browser & Device Support

The W3C Geolocation API works for the following desktop and mobile browsers:

Geolocation API Supported Browsers

Platform	Browsers
desktop	Firefox from version 3.5, Opera 10.6, Google Chrome, Internet Explorer 9.0, Safari 5
mobile	Android (firmware 2.0 +), iOS, Windows Phone

JavaScript Implementation

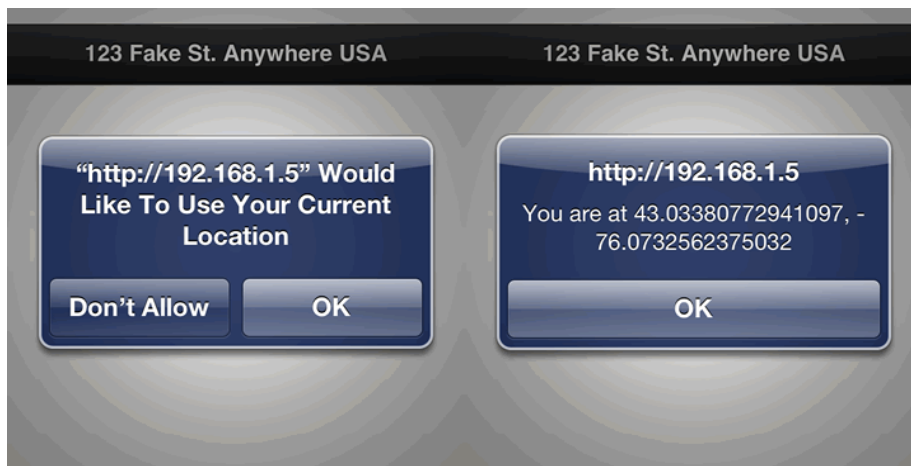
For devices that support the GeoLocation API, no external JavaScript file needs to be linked - the JavaScript functionality is inherent in the browser itself. The core of the code is the geolocation object, a child of the navigator object:

```
if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition(successFunction, failureFunction, {enableHighAccuracy:false, maximumAge:30000, timeout:27000});
}
```

The `if` statement tests if the `navigator.geolocation` object exists on this device/browser. If so, we call the object's `getCurrentPosition` function - the first two parameters refer to a function (which we've called `successFunction`, but we could have used any name) to call when `getCurrentPosition` is successful in finding the user's location, and a function (`failureFunction`, in this example) to call when `getCurrentPosition` was not successful in finding the user's location; this parameter is optional. Both `failureFunction` and `successFunction` are functions we will write to handle the success and failure cases. The third parameter, a JSON-formatted hash of options, allows fine control over implementation details; we will omit this optional parameter when calling `getCurrentPosition`.

A Simple Example

Let's look at a simple page (IntegrationPhone/Demos/mapping.html) that pops up a JavaScript message with the user's latitude and longitude location. Users who visit our page will be prompted to explicitly allow our page to use their current location; see the left screenshot below. If they accept, our page will respond with their latitude and longitude, as on the right.



Code Sample:

[IntegrationPhone/Demos/mapping.html](#)



We test, with an `if` statement, for the presence of the `navigator.geolocation` object; if so, we call the `getCurrentPosition` function. We write two functions, `locateSuccess` and `locateFail`, to handle the success and failure cases, respectively.

On successful finding of the user's location, function `locateSuccess` is invoked. Note that, as discussed above, we can name this function anything we like (though, of course, a meaningful name containing "success" makes sense here), as long as the name of the function is the same name we use for the first parameter of the `getCurrentPosition` function. The `loc` parameter exposes a variety of information about the user's location; we assign `loc.coords.latitude` and `loc.coords.longitude` to two local variables, and show those values to the user in a JavaScript alert (popup) window.

If the user's location is not found, function `locateFail` is invoked. We use a JavaScript `switch` statement to evaluate parameter `geoPositionError`'s `code` property: if 0, something unknown happened; if 1, the user denied permission; etc.

Of course, we could extend this application extensively: we might use Google's [Places](#) library to allow the user to perform a search for "bookstores" and to return human-friendly addresses (hopefully Nan & Bob's!), rather than latitude and longitude. Check out the Google Maps [Demo Gallery](#) for some inspiration and code samples.

The parameter passed to the success function, which we named `loc` in our example above, exposes a useful set of properties:

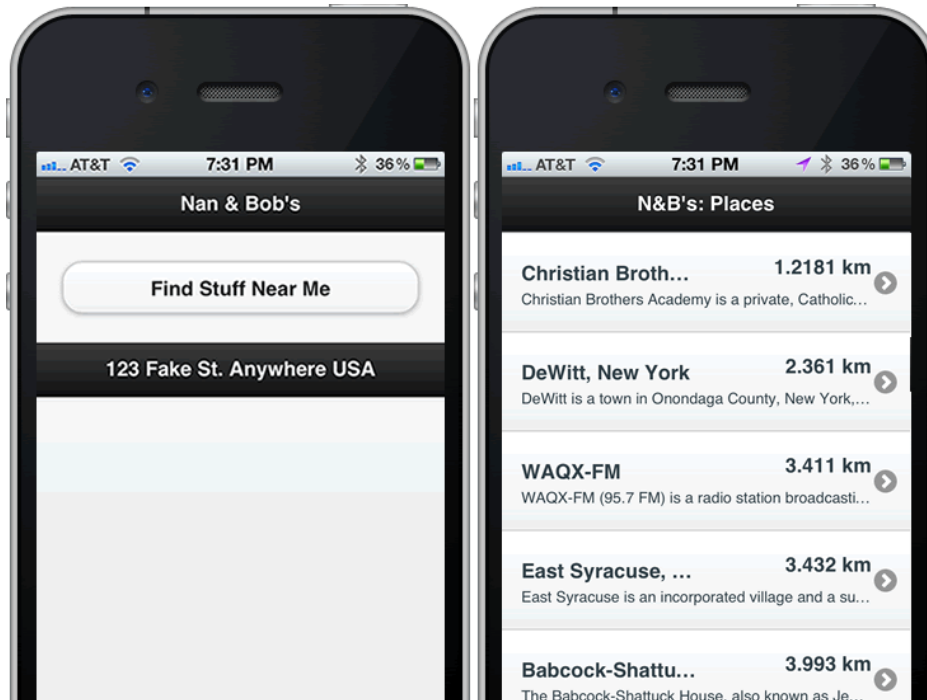
Position Properties

Property	Units
<code>coords.latitude</code>	degrees
<code>coords.longitude</code>	degrees

coords.altitude	meters (may be null)
coords.accuracy	meters
coords.altitudeAccuracy	meters (may be null)
coords.heading	degrees clockwise (may be null)
coords.speed	meters/second (may be null)
timestamp	a date and time

Finding Nearby Places

In this example, we'll make use of the free GeoNames service - specifically their [Find nearby Wikipedia Entries](#) service. We'll find the user's location as we did above, pass the latitude and longitude to GeoNames, and receive back a JSON-formatted list of nearby places' Wikipedia entries, which we use to populate a `listview`

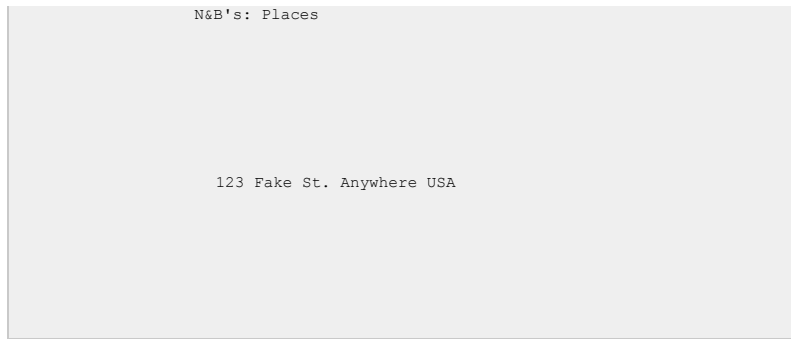


We can imagine that Nan & Bob, being community-minded folks, might want to allow visitors to their new site to find information about nearby places of interest - we'll add a button to the first page of this demo that allows users to find their current location then to see a short list of nearby points of interest. Users can click to read more on Wikipedia. Open up [IntegrationPhone/Demos/places.html](#) to test it out and to view the code.

Code Sample:

[IntegrationPhone/Demos/places.html](#)





This site comprises two pages: the home page and a page to display the returned results as a listview. Note the empty `listview` (an unordered list with id `placelist`) on the `#places` page.

We add a jQuery event handler to listen for clicks on the "Find Stuff Near Me" button - the button with id `findme`. When clicked, we call `navigator.geolocation.getCurrentPosition` which, if successful (the device supports it, user has approved the request to find his/her location, a location was returned), calls function `locateSuccess`. The `locateSuccess` function, in turn, calls function `geonames.search`.

Function `geonames.search`, which we define as part of namespace `geonames`, uses the jQuery function `$.getJSON` to retrieve a JSON-formatted list of places from GeoNames by passing a correctly formatted URL (with values for `lat` and `lng`, as defined by GeoNames, as well as parameters `style`, `radius`, and `maxrows`) - we found all of the details for this from the [GeoNames website](#).

The next block of code iterates over the results returned from `$.getJSON`: we create an empty list item, append to it the data for each place record, and append the constructed list item to the `placelist` listview. We refresh the listview and change to the `#places` page for the user to view the results. Note that we use here the jQuery core iterator `$.each` - useful for looping over any sort of JavaScript collection, in this case a JSON result set. See the jQuery docs for more information on [\\$.each](#)

Finding Weather Data

Nan & Bob would like to offer visitors to their website the opportunity to view weather conditions where they are. (We suggested showing the weather at the store, rather than the website-visitor's location - but hey, it's their site.) In the next exercise, you will make use of another GeoNames service (`findNearByWeatherJSON`) that returns the local weather conditions when supplied a latitude and longitude.

You'll need to use the jQuery iterator `each`:

```
$.each(obj, function(i, val) {
    alert(i + ' - ' + val);
});
```

If `obj` were a valid object (like the one that will be returned by the `findNearByWeatherJSON` service), then the code above would generate an alert for each name/value pair, with `i` as the field name and `val` as the field value. Note that this is slightly different from the earlier example: here we are iterating over the fields of the object, whereas before we were iterating over multiple objects.

Lesson 1, Activity 7: Retrieving Weather Data

Duration: 25 to 35 minutes.

In this exercise, you will allow visitors to the Nan & Bob's site to find weather conditions for their current location. The result will look something like this:



1. Open [IntegrationPhone/Exercises/index.html](#) in a file editor.
2. Add a button to the home page with id `findweather` and label "Local Weather".
3. Note that a new page (`#weather`) has been added for you; it contains a `listview` with id `weatherconditions`.
4. Using the demo above as an example, write code to call the Geolocation API `getCurrentPosition` function when the user clicks the button.
5. On successful retrieval of the user's location, use the `$.getJSON` function to retrieve weather data; the URL supplied should be of the format <http://ws.geonames.org/findNearByWeatherJSON?lat=43.048122&lng=-76.147424> but, of course, with the user's latitude and longitude.)
6. GeoNames will return an object with the weather data - use `$.each` to iterate over the fields and add each name/value pair to the `weatherconditions` listview.
7. Change the page to `#weather` and refresh the `#weatherconditions` listview.
8. Be sure to test your code in a mobile browser.

Solution:[IntegrationPhone/Solutions/index.html](#)

```

---- CODE OMITTED ----

<script>
    $(document).on('pageinit', function() {
        setContent(window.orientation);

        //listen for clicks on the "Local Weather" button:
        $('#findweather').click(function() {
            if (navigator.geolocation) {
                navigator.geolocation.getCurrentPosition(locateSuccess, locateFail);
            }
        });
    });
    function locateSuccess(loc) {
        weatherObservation.search(loc.coords.latitude, loc.coords.longitude);
    }

```



```

    }
    function locateFail(geoPositionError) {
        switch (geoPositionError.code) {
            case 0: // UNKNOWN_ERROR
                alert('An unknown error occurred, sorry');
                break;
            case 1: // PERMISSION_DENIED
                alert('Permission to use Geolocation was denied');
                break;
            case 2: // POSITION_UNAVAILABLE
                alert('Couldn\'t find you...');
                break;
            case 3: // TIMEOUT
                alert('The Geolocation request took too long and timed out');
                break;
            default:
        }
    }
}
var weatherObservation = {};
weatherObservation.search = function(lat,lng) {
    // retrieve data from Geonames in JSON format
    $.getJSON('http://ws.geonames.org/findNearByWeatherJSON?lat=' + lat + '&lng=' + lng, function(data) {
        // iterate over results and add to DOM
        $.each(data.weatherObservation, function(i, val) {
            $('<li></li>')
                .hide()
                .append(i + ': ' + val)
                .appendTo('#weatherconditions')
                .show();
        });
        // change to "Weather" page
        $.mobile.changePage('#weather');

        // refresh the list
        $('#weatherconditions').listview('refresh');
    });
};
};

---- C O D E   O M I T T E D ----

<div data-role="page" id="weather">
    <div data-role="header">
        <h2>N&B: Weather</h2>
    </div>
    <div data-role="content">
        <h3>Weather where you are:</h3>
        <ul data-inset="true" data-role="listview" id="weatherconditions"></ul>
    </div>
    <div data-role="footer">
        <div data-role="navbar">
            <ul>
                <li><a href="#home">Home</a></li>
                <li><a href="#books">Books</a></li>
                <li><a href="#eats">Eats</a></li>
            </ul>
        </div>
        <h3>
            123 Fake St. Anywhere USA
        </h3>
    </div>
</div>
</body>
</html>

```

We add a handler to listen for clicks on the #findweather button, calling navigator.geolocation.getCurrentPosition. A successful geolocation calls weatherObservation.search.

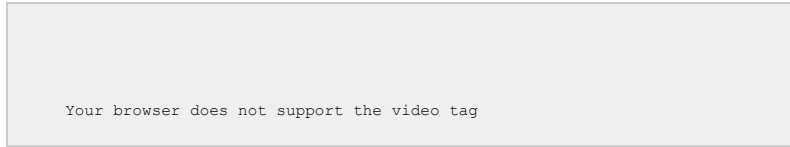
The search function requests JSON data from GeoNames, sending the user's latitude and longitude as part of the request URL. We iterate over the object returned to populate the#weatherconditionslistview with the results.

Lesson 1, Activity 9: Video

The ease of embedding video from sites like YouTube or Vimeo is great - but embedding isn't always a perfect option. Worries over intellectual property rights (others can embed my videos), loss of traffic (visitors view my videos on YouTube instead of my site), professionalism (worries that visitors might think my site less professional if videos are embedded), and other concerns might dictate that you host your videos on your own site.

The HTML5 `video` tag, along with appropriate choices for video file format, makes relatively easy the process of presenting videos to be playable regardless of the viewing device.

First, let's look at the video tag:



The `width` and `height` attributes set the width and height, of course; we'll also set the CSS `max-width` of the video element to ensure a responsive handling of width. The presence of the `controls` attribute dictates that the player will show controls ("play", "pause", etc.) The `poster` attribute is the image to show on the player before video playing starts.

The `video` tag encloses a series of `source` tags, which specify alternative video files from which the browser may choose, depending on the media types or codecs the browser supports.

In the example above, we offer browsers three options to pick from: `.mp4`, `.webm`, and `.ogg` versions of our video.

Last, the content inside the video tag but outside the source tags is what users will see if their browser does not support the video tag.

Video Formats

Specifics about video file formats is a bit beyond the scope of this course. What we really need is a list of best formats to use and some information on how to convert to those formats.

Flash video, often the format in which embedded video is presented for desktop browsers, generally won't work on mobile devices. Instead, we'll concentrate on three formats:

Video File Formats

Format	File Extension	Notes
MPEG 4	.mp4	Based on Apple older QuickTime format; supported natively in all browsers.
Ogg	.ogv, .ogg	Open source; unencumbered by patents; supported natively by Firefox >3.5, Opera >10.50, Chrome >3.0.
WebM	.webm	New (2010) format; supported natively by Chrome, Firefox, and Opera.

So to publish video - specifically, video able to be viewed on different desktop and smartphone browsers - we do the following:

1. Take our original video file (the raw file from the camera).
2. Convert it to each of the file formats above.
3. Publish with the `video` tag.

The user sees a video player with their browser - whether it be Safari on an iPhone, Internet Explorer on a Windows PC desktop, or anything else - picking the video format that works best.

Further Reading

[Dive Into HTML5](#) is a great resource for more information on HTML5 in general; [Video on the Web](#) is, in particular, an excellent source of information on creating, saving, and publishing video in HTML5 and, thus, for mobile viewing.