Learn by doing: less theory, more results

PhoneGap
Third Edition

Create, develop, debug, and deploy your very own mobile applications with PhoneGap

Foreword by Ranga Srinivasan, CTO & President, Ameex Technologies

Beginner's Guide

Purusothaman Ramanujam   Giorgio Natili
In this package, you will find:

- The authors biography
- A preview chapter from the book, Chapter 7 'Accessing the Device Sensors and Locations API'
- A synopsis of the book’s content
About the Authors

Purusothaman Ramanujam is a mainframe consultant and an automation expert by profession and a full stack web developer and trainer by passion. He holds a bachelor’s degree in information technology and a master’s degree in financial management. He has a strong domain knowledge of financial markets. Apart from working full-time at a multinational corporation, he freelances and contributes to various open source projects. In his personal blog at http://www.purusothaman.me, he writes about technology, Tamil poetry, French, and more.

Giorgio Natili is an author, educator, community leader, and lead UI engineer at McGraw Hill Education; he is also a publisher of print and digital information services for the academic, professional, and library markets.

At McGraw Hill, he is involved with multiple client-side teams on cross-platform mobile and web applications, developing custom components (web and native) and bringing mock-ups to life!

Giorgio was also the founder of GNStudio, a boutique Rome-based development and design studio specializing in engaging and accessible web and mobile experiences.

As a strong proponent of agile development practices, Giorgio’s areas of expertise include web standards-based application development, frontend development, gaming, video streaming, iOS development using Swift and Objective-C, Android development, and hybrid development (mobile and wearables).

Giorgio is the founder of Mobile Tea (http://www.meetup.com/mobiletea/), a fresh and innovative approach to community tech events, and is involved in several other community-driven events in Italy and the US.
Preface

*PhoneGap Beginner’s Guide Third Edition* will help you break into the world of mobile application development. You will learn how to set up and configure your mobile development environment, implement the most common features of modern mobile apps, and build rich, native-style experiences. All the examples deal with real use case scenarios covering the functionality of various plugins.

What this book covers

Chapter 1, *Introduction to PhoneGap*, covers how to set up dependencies and mobile platform SDKs in your development environment.

Chapter 2, *Building Your First PhoneGap Project*, deals with PhoneGap internals, project structure, and using CLI tools.

Chapter 3, *Mobile Frameworks*, gives an introduction to various mobile frameworks and sample projects using jQuery Mobile.

Chapter 4, *Working with Plugins*, covers information about PhoneGap plugins and using them in the application.

Chapter 5, *Using Device Storage and the Files API*, deals with offline data storage capabilities and the Files plugin.

Chapter 6, *Using the Contacts and Camera APIs*, covers how to implement the Contacts API and how to interact with the device camera by using the Camera API.

Chapter 7, *Accessing the Device Sensors and Locations API*, deals with using device sensors and the Locations API and their power and limitations with respect to the plugins provided by PhoneGap.
Chapter 8, Advanced PhoneGap, covers some advanced topics such as adding multilanguage and touch gesture support to your application.

Chapter 9, Getting Ready for Release, helps you understand how to make your application ready for public release in various application stores.

Chapter 10, A Sample PhoneGap Project, deals with the development of a full-fledged PhoneGap application from scratch, using the most common PhoneGap plugins.

Appendix A, The JavaScript Quick Cheat Sheet, is a mini cheat sheet for commonly used JavaScript methods and properties.

Appendix B, Publishing Your App, describes how you can publish your apps on different app stores.

Appendix C, Related Plugin Resources, is a list of some related plugins that can be used with PhoneGap.

Appendix D, PhoneGap Tools, describes tools that will help you to debug and test your applications without any difficulties.
The use of device sensors opens the doors to sophisticated apps, which may improve user experience and enhance the capabilities of a modern app. It’s very important for a mobile developer to understand the power and limitations of device sensors in order to effectively use the APIs provided by the PhoneGap framework. Location data allows a mobile developer to tag every piece of information with the device’s position. In this chapter, you will also learn to couple the Location API with your app.

In this chapter, you will:

- Learn which are the most common device sensors and how to use them in order to enhance the user experience
- Get an overview of the device orientation and device motion events using the accelerometer
- Learn how to work with device sensors directly with JavaScript
- Learn how to use the Compass API of PhoneGap
- Learn about geolocation and how its data is available in the device
- Learn how to use the PhoneGap Geolocation API and how to integrate the Google Maps API in an app

Location data allows a mobile developer to tag every piece of information with the device’s position. This kind of meat tagging enables the use of very contextualized apps. The PhoneGap framework provides a Geolocation API that is simple to use, easy to understand, and very powerful.
Introducing device sensors

Humans have senses (touch, hear, smell, and so on); a phone has digital "senses": touch, geolocation, orientation, and motion. A sensor is a device component that measures a physical quantity and converts it into a signal that is understandable to software. Modern mobile phones come with a variety of sensors that can support users when completing their daily tasks. By tapping into a device sensor, you can enhance the end user experience and develop sophisticated apps.

Sensors can be hardware-based or software-based. Hardware-based sensors are physical components built into a handset or tablet device. They derive their data by directly measuring specific environmental properties such as acceleration, geomagnetic field strength, or angular change. Software-based sensors are not physical devices, although they mimic hardware-based sensors.

Typical device sensors are the accelerometer, gyroscope, compass, barometer, orientation sensor, and so on.

Not all devices, nor their operating systems support the same sensors, so you have to know which devices you want to target before considering which sensors to use in your app. The device sensors typically are divided into the following categories:

- Motion sensors
- Environmental sensors
- Position sensors

The motion sensors measure acceleration forces and rotational forces along the three axes. Hardware parts such as the accelerometer, gravity sensors, gyroscopes, and rotational vector sensors belong to this category. The environmental sensors measure various environmental parameters such as ambient air temperature and pressure, illumination, and humidity. The barometers, photometers, and thermometers belong to this category of sensors. The position sensors measure the physical position of a device.

As already mentioned, each operating system offers different sensors. From a developer's point of view, this means that to work on different platforms, you have to understand how sensors work on each one. When working with PhoneGap, you can safely use the Accelerometer and Compass APIs across different platforms. Furthermore, you can rely on the onboard browser capabilities to get additional sensor information such as the device orientation.

The accelerometer is actually made up of three accelerometers and each one measures the changes in velocity (that is, linear acceleration) overtime along the linear path on the axes x, y, and z. Combining the data of the three accelerometers, you can get the device movement and orientation.
The gyroscope is always part of the motion sensors and it measures the rate of rotation around the three axes, usually roll, pitch, and yaw.

The magnetometer measures the strength of the magnetic field surrounding the device and in the absence of any strong local fields, these measurements will refer to the magnetic field of the Earth. In this way, the device is able to determine its heading with respect to the geomagnetic North Pole; using the heading values, it's possible to determine the yaw of the device too. Magnetic heading updates are available even if the user has switched off location updates in the settings application; the reported values are positive numbers from 0 to 360. The real heading of the user, when they are holding the device in landscape mode, is the reported heading plus 90 degrees.

The iOS platform provides all the common sensors a developer can expect such as accelerometer, magnetometer, gyroscope, and the proximity sensor.

The Android platform provides four additional sensors that allows you to monitor various environmental properties: ambient humidity, luminance, ambient pressure, and ambient temperature. All the sensors are hardware-based and are available only if a manufacturer has built them into a device.

![You can find a complete demo of the Android sensors on the Google Play store; just search and install the Android Sensor Box app.]

The Windows Phone 7.5/8 platform offers wide support for sensors. You can use the Inclinometer sensor to detect the pitch, roll, and yaw of the device or you can create complex 3D apps using the Quaternion sensor (quaternion is the quotient of two directed lines in a three-dimensional space). For a complete overview of the Windows Phone sensor APIs, please refer to the online documentation at http://msdn.microsoft.com/library/windows/apps/windows.devices.sensors.

The location capabilities of a device rely on several sensors called position sensors. Devices normally use multiple positioning methods to provide different granularities of location data. The sources of position data vary in terms of accuracy, startup time, and power signature, and include the following:

- GPS
- A-GPS
- Cell tower triangulation
- Wi-Fi triangulation
- IP address
Accessing the Device Sensors and Locations API

With the continuous evolution of sensors, end user expectations are growing and the quality of the apps available on the market is increasing.

Lapka Electronics released a set of sensors and an app that is able to translate environmental data to read values easily. Using their sensors and app, you can measure electromagnetic pollution, humidity, amounts of nitrates in raw foods and drinking water, and so on. More information about these sensors is available online at http://mylapka.com/.

Sensors and human-computer interaction

Sensors evolved and are still evolving very fast and are influencing how creative people are designing apps. The new generation of apps rely on voice commands, gestures, and more in order to allow the user to control apps in a more intuitive way. An app is now able to perceive the user’s intentions based on the sensor data it collects. The use of sensors to make apps (and computers) more intuitive to control is known as perceptual computing. This initiative is led by Intel and has various applications including video conferencing, gaming, and so on.

By contrast, augmented reality is about extending how humans interface with the physical world through computers. Using an augmented reality interface, you can add additional information to the external environment and create amazing and useful apps. On mobile devices, the implementation of an augmented reality application heavily depends on the sensors on the devices, such as video cameras and orientation sensor.

A nice example of the kind of interactions you can reach through sensors is an app for iOS named Car Finder. The app stores the position of a car when the user takes a picture of it and then provides to the user the information needed to find where he/she had parked the car.

The capability to use sensors and the data they return is increasingly paramount for mobile developers. The sensors supported by PhoneGap are limited, but PhoneGap is simply a wrapper that makes it easier to separate your presentation layer from the native device code. For this reason, you can start to write additional native code around the PhoneGap wrapper to extend its capabilities.

An interesting resource on sensor development is available on the Microsoft website at http://research.microsoft.com/en-us/groups/sendev/, where you can find papers and resources to help you get started with sensors.
Accelerometer

The PhoneGap Accelerometer API allows you to detect the device movement change values relative to device orientation. Note that the accelerometer detects the values as a delta movement relative to the current device position. Even more important, it takes into consideration the effect of gravity (that is, $9.81 \text{ m/s}^2$), so that when a device is lying flat on a table facing up, the value returned should be $x = 0$, $y = 0$, and $z = 9.81$.

As with any other plugin, you have to install the plugin before you can use it in your project. The plugin can be added to your project using the following command:

$ cordova plugin add cordova-plugin-device-motion

Once the plugin is installed on the project, a `navigator.Accelerometer` global object, is created and it is available to use once the `deviceready` event is fired. However, it is recommended that you still check for its presence before using it:

```javascript
function onDeviceReady() {
    if( typeof navigator.accelerometer === "undefined"){
        //plugin is ready now
    }
}
```

You can detect the device acceleration data using the `getCurrentAcceleration` method or by setting up a watcher through the `watchAcceleration` method. Both methods are available on the `navigator.accelerometer` object and accept similar arguments.

The `getCurrentAcceleration` method accepts a success and a failure callback function as an argument and doesn’t return anything. The `watchAcceleration` method accepts an additional argument in order to define the options and return a reference to the current watcher.

In order to constantly watch the acceleration data, you have to define the frequency at which you want to recover data and store the value returned by the `watchAcceleration` method in a variable:

```javascript
var options = {frequency: 300};
var currentAcceleration = navigator.accelerator.watchAcceleration
    (onSuccess, onFailure, options);
```
Accessing the Device Sensors and Locations API

The onSuccess handler receives an Acceleration object as an argument, accessing its property and making it possible to read the acceleration on each axis:

```javascript
function onSuccess(acceleration) {
    console.log('Acceleration X: ' + acceleration.x );
    console.log('Acceleration Y: ' + acceleration.y);
    console.log('Acceleration Z: ' + acceleration.z );
}
```

The failure handler doesn’t receive any argument, but it’s pretty useful to handle possible errors when accessing the device’s accelerometer:

```javascript
function onError() {
    console.log('Error accessing the accelerometer');
}
```

In order to stop watching the accelerometer data, it’s sufficient to call the clearWatch method defined on the accelerator object, passing the reference to the variable previously used to store the result of the watchAcceleration method:

```javascript
navigator.accelerometer.clearWatch(currentAcceleration);
```

This method doesn’t accept any additional handler.

All the sensor APIs of PhoneGap work in a similar way; you will always have to use a getCurrentSENSOR and a watchSENSOR method (where SENSOR is the name of the sensor) to obtain data from the sensor. In order to stop watching a sensor, you will always use the clearWatch method.

**Detecting shakes**

Using the information recovered from the accelerometer API, it’s possible to understand whether the user is shaking the device.
Device orientation events

The cordova-plugin-device-motion plugin only supports access to the acceleration information. In order to handle the orientation changes, you have to rely on the JavaScript APIs of the target platform browser. When you want to update the user interface when the device orientation changes, you have to use CSS media queries; any other business logic can be handled using JavaScript due to the fact that PhoneGap uses the web view to render the app user interface.

Using JavaScript, you can set up a listener for the `orientationchange` event and another listener for the `deviceorientation` event in order to handle the device orientation. The first event is fired each time the orientation of the device changes; the second event is fired when the physical orientation of the device changes. Both the listeners have to be registered to the `window` object:

```javascript
window.addEventListener('orientationchange', EVENT_HANDLER);
window.addEventListener('deviceorientation', EVENT_HANDLER);
```

The `orientationchange` event handler is commonly used to detect the screen orientation after it has changed. Once the orientation changes, the app receives a notification for several events. The following table summarizes these events and the orientation property value:

<table>
<thead>
<tr>
<th>Device and user gesture</th>
<th>Events fired</th>
<th>Orientation</th>
</tr>
</thead>
<tbody>
<tr>
<td>iPad to landscape</td>
<td>resize</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>orientationchange</td>
<td>90</td>
</tr>
<tr>
<td>iPad to portrait</td>
<td>resize</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>orientationchange</td>
<td>0</td>
</tr>
<tr>
<td>iPhone to landscape</td>
<td>resize</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>orientationchange</td>
<td>90</td>
</tr>
<tr>
<td>iPhone to portrait</td>
<td>resize</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>orientationchange</td>
<td>0</td>
</tr>
<tr>
<td>Android phone to landscape</td>
<td>orientationchange</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>resize</td>
<td>90</td>
</tr>
<tr>
<td>Android phone to portrait</td>
<td>orientationchange</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>resize</td>
<td>0</td>
</tr>
</tbody>
</table>
The `deviceorientation` event is very powerful. It returns to the handler an instance of the `DeviceOrientationEvent` event with the following information:

- **alpha**: This returns the rotation of the device around the z axis
- **beta**: This returns the rotation of the device around the x axis
- **gamma**: This returns the rotation of the device around the y axis

In order to improve the performance of your app, consider using the `event-handler` function to do no more than save current values from the sensor data into variables. Then, move your calculations or DOM manipulations into a new function executed at a fixed time.

**Handling orientation with JavaScript**

It's time to put into practice what you have learned about the device orientation events. Let's work on a very basic sample that is able to show the screen orientation in a `div` element rotated according to the device's physical orientation.

**Time for action – handling device orientation with JavaScript**

Execute the following steps:

1. Open the command-line tool and create a new PhoneGap project named `orientationevents`, and add the platforms you want to target for this example.
2. Install the plugin to your project:
   ```bash
   $ phonegap plugin add cordova-plugin-device-motion
   ```
3. Go to the `www` folder, open the `index.html` file, and add `div` with the `#orientation` ID inside the main `div` of the app beneath `#deviceready`:
   ```html
   <div class="app">
     <h1>Apache Cordova</h1>
     <div id="deviceready">
       ......
     </div>
     <div id="orientation">
     </div>
   </div>
   ```
4. Go to the css folder and define two new rules inside the index.css file to give a border and a bigger font size to div and its content. You can even add the CSS styles directly to the head section of your HTML page:

```css
#orientation{
    width: 230px;
    border: 1px solid rgb(10, 1, 1);
}

#orientation p{
    font-size: 36px;
    font-weight: bold;
    text-align: center;
}
```

5. Go to the js folder, open the index.js file, and define a new function to easily detect whether the device can handle the orientationchange and deviceorientation events. Alternatively, you can even have the script embedded in your HTML page:

```javascript
orientationSupported: function(){
    try {
        return 'DeviceOrientationEvent' in window &&
        window['DeviceOrientationEvent'] !== null;
    } catch (e) {
        return false;
    }
}
```

6. In the deviceready function, add two listeners if the device supports the orientationchange and deviceorientation events:

```javascript
if(orientationSupported){
    window.addEventListener('orientationchange', orientationChanged);
    window.addEventListener('deviceorientation', updateOrientation);
}
Accessing the Device Sensors and Locations API

```javascript
else{
    alert('Orientation not supported!');
}

7. Define the `orientationChanged` event handler and use it to print the current device orientation on screen:
   ```javascript
   orientationChanged: function() {
       var element = document.querySelector('#orientation');
       element.innerHTML = '<p>' + window.orientation + '</p>';
   }
   ```

8. Define the handler for the `deviceorientation` event and use the information provided by the device's sensor to change the 3D transformation of the `div orientation`:
   ```javascript
   updateOrientation: function(event) {
       var alpha = event.alpha,
           beta = event.beta,
           gamma = event.gamma;

       var element = document.querySelector('#orientation');
       var rotation = 'rotateZ(' + alpha + 'deg) rotate(' + beta + 'deg) rotateY(' + gamma + 'deg)';
       // For brevity the browser prefixes have been removed
       element.style.transform = rotation;
   }
   ```

9. Open the command-line tool again, locate the main project folder, and then compile the app and test it on every platform you previously added.

Here is the complete code of this example:

```html
<!DOCTYPE html>
<html>
    <head>
        <!--Other section removed for sake of simplicity -->
        <title>Media Capture Example</title>
    </head>
    <body>
        <!-- Code for accessing device sensors and locations -->
    </body>
</html>
```
<div class="app">
<h1>Apache Cordova</h1>
<div id="deviceready"></div>
<div id="orientation"></div>
</div>
<script type="text/javascript" src="cordova.js"></script>
<script type="text/javascript">
document.addEventListener("deviceready", onDeviceReady, false);

function onDeviceReady() {
  if(orientationSupported){
    window.addEventListener('orientationchange', orientationChanged);
    window.addEventListener('deviceorientation', updateOrientation);
  }else{
    alert('Orientation not supported!');
  }
}

orientationSupported: function(){
  try {
    return 'DeviceOrientationEvent' in window &
    window['DeviceOrientationEvent'] !== null;
  } catch (e) {
    return false;
  }
}
orientationChanged: function(){
  var element = document.querySelector('#orientation');
  element.innerHTML = '<p>' + window.orientation + '</p>';
}

updateOrientation: function(event){
  var alpha = event.alpha,
  var beta = event.beta,
  var gamma = event.gamma;
  var element = document.querySelector('#orientation');
  var rotation = 'rotateZ(' + alpha + 'deg) rotate(' + beta +
  'deg) rotateY(' + gamma + 'deg)';
  // For brevity the browser prefixes have been removed
  element.style.transform = rotation;
}

</script>
</body>
</html>

What just happened?
You handled the orientation events using JavaScript and deployed the result to a device using PhoneGap. The app is able to get the device screen orientation and the current position in real time.

Compass
The PhoneGap Compass API allows you to obtain the direction that the device is pointing to. The compass is a sensor that detects the direction or heading in which the device is pointed and returns the heading of the device in degrees using values from 0 to 359.99. The Compass API works similarly to the Accelerometer API; in fact, you can read the current device heading or you can define a watcher in order to continuously read the heading value.

The Compass API is available on the compass property of the navigator object and exposes the following functions:

- compass.getCurrentHeading: This reads the current compass heading through a handler
compass.watchHeading: This reads the compass heading at a specific time interval through a handler and returns a reference to it

compass.clearWatch: This stops a previously defined time interval reading handler

The getCurrentHeading and watchHeading functions accept very similar arguments; the only difference is the last argument of the watchHeading function that allows you to configure it. In order to read the current heading of the device, it is sufficient to execute the getCurrentHeading function, specifying a success and an error handler:

```plaintext
navigator.compass.getCurrentHeading(onSuccess, onError);
```

The `onSuccess` handler receives as an argument a `CompassHeading` object with the following properties:

- **magneticHeading**: This is the heading in degrees from 0 to 359.99
- **trueHeading**: This is the heading relative to the geographic North Pole in degrees
- **headingAccuracy**: This is the deviation between the reported heading and the true heading in degrees
- **timestamp**: This is the time at which this heading was determined

The error handler receives a `CompassError` object as an argument; the `CompassError` object has a property named `code` that returns two possible values such as `CompassError.COMPASS_INTERNAL_ERR` or `CompassError.COMPASS_NOT_SUPPORTED`:

```plaintext
function onError (error) {
    switch(true){
        case error.code == CompassError.COMPASS_INTERNAL_ERR:
            navigator.notification.alert('Compass Error!', null, 'Info', 'OK');
            break;

        case error.code == CompassError.COMPASS_NOT_SUPPORTED:
            navigator.notification.alert('Compass Unavailable!', null, 'Info', 'OK');
            break;

        default:
            navigator.notification.alert('Generic Error!', null, 'Info', 'OK');
    }
}
```
Accessing the Device Sensors and Locations API

The `watchHeading` function works like the `getCurrentHeading` function. The only difference is that it accepts an additional `CompassOption` object that allows you to set up how often to retrieve the compass heading in milliseconds (that is, frequency) and the change in degrees required to initiate the success handler (that is, filter):

```javascript
var options = {frequency: 300};
var currentHeading = navigator.compass.watchHeading(
onSuccess, onError, options);
```

In order to stop watching the heading value changes, it is sufficient to use the `clearWatch` function and the reference to the current heading watcher:

```javascript
clearWatch(currentHeading);
```

The `trueHeading` property of the `CompassHeading` object is not supported on Android. It returns the same value as `magneticHeading`, and the `headingAccuracy` value will always be 0 as there is no difference between `magneticHeading` and `trueHeading`. On iOS, the `trueHeading` property is returned only when location services are running using the `watchLocation` function.

Creating a compass

Reading the current heading of a device is a common task for a developer in several use cases such as traffic apps, augmented reality apps, or any app that incorporates a sense of direction. Let's see how to create a complete compass with PhoneGap:

The images used to render the compass are available under the Creative Commons license at [http://commons.wikimedia.org/wiki/File:Compass.svg](http://commons.wikimedia.org/wiki/File:Compass.svg). Before starting to work on this example, download the image and create three separate PNG files for the background, dial, and arrow. As it's a SVG vector file, you can handle each layer of the image and edit it as your wish. To edit the image, you can use any vector editing applications such as Adobe Illustrator or a free application such as Inkscape.
Chapter 7

**Time for action – using the Compass API**

Execute the following steps:

1. Open the command-line tool, create a new PhoneGap project named `compass`, and add the platforms you want to target for this example.

2. Add the Compass API plugin using the following command:
   ```bash
   $ cordova plugin add cordova-plugin-device-orientation
   ```

3. Go to the `www` folder and open the `index.html` file. The three `div` tags are used to handle the compass arrows and the background:
   ```html
   <section id="compass">
   <div id="compassbg"></div>
   <div id="north"></div>
   <div id="arrow"></div>
   </section>
   ```

4. Go to the `css` folder, open the `index.css` file, and define the rules needed to have a separate background for each element of the compass:
   ```css
   #compassbg {
      background-image: url(../img/Compass.png);
   }
   #north {
      background-image: url(../img/arrow_direction.png);
   }
   #arrow {
      background-image: url(../img/arrow_beta.png);
   }

   #compass, #arrow, #north, #compassbg {
      background-repeat: no-repeat;
      background-size: cover;
      position: fixed;
      width: 286px;
      height: 286px;
   }
   ```
5. Go to the js folder, open the index.js file, and add a new variable in order to store a reference to the watcher that you will define to monitor the device heading. Alternatively, you can have the scripts embedded directly in the page:

```javascript
var currentHeading = null;
```

6. Locate the deviceready function and add inside it the snippet of code needed in order to check the device heading every 150 milliseconds:

```javascript
var options = {frequency: 150};
currentHeading = navigator.compass.watchHeading
  (onCompassSuccess, onCompassError, options);
```

7. Create a new function named onCompassSuccess and inside its body, start to read the heading data stored in the received argument; use it to rotate the compass elements:

```javascript
function onCompassSuccess(heading) {
  var magneticHeading = heading.magneticHeading;
  var trueHeading = heading.trueHeading;

  var compass = document.querySelector('#compassbg');
  var north = document.querySelector('#north');

  var compassRotation = 'rotate(' + magneticHeading + 'deg)';
  var northRotation = 'rotate(' + trueHeading + 'deg)';
  var compassSytle = compass.style;
  var northStyle = north.style;

  compassSytle.transform = compassRotation;
  northStyle.transform = northRotation;
}
```

8. Define the function to capture the failures, if any:

```javascript
function onCompassError(error) {
  alert("Error with Compass");
}
```

9. Open the command-line tool again, locate the main project folder, compile the app, and test it on every platform you previously added.

For the example, we just saw, you can find the complete code here:

```html
<!DOCTYPE html>
<html>
<head>

```
<!-Other section removed for sake of simplicity-->
<title>Media Capture Example</title>
<style>
#compassbg {
  background-image: url(../img/Compass.png);
}
#north {
  background-image: url(../img/arrow_direction.png);
}

#arrow {
  background-image: url(../img/arrow_beta.png);
}

#compass, #arrow, #north, #compassbg {
  background-repeat: no-repeat;
  background-size: cover;
  position: fixed;
  width: 286px;
  height: 286px;
}
</style>

</head>
<body>

<section id="compass">
  <div id="compassbg"></div>
  <div id="north"></div>
  <div id="arrow"></div>
</section>

<script type="text/javascript" src="cordova.js"></script>

<script type="text/javascript">
  var currentHeading = null;

  document.addEventListener("deviceready", onDeviceReady, false);

  function onDeviceReady() {

```
Accessing the Device Sensors and Locations API

```javascript
var options = {frequency: 150};

currentHeading = navigator.compass.watchHeading(onCompassSuccess, onCompassError, options);
}

function onCompassSuccess(heading){
    var magneticHeading = heading.magneticHeading;
    var trueHeading = heading.trueHeading;

    var compass = document.querySelector('#compassbg');
    var north = document.querySelector('#north');

    var compassRotation = 'rotate(' + magneticHeading + 'deg)';
    var northRotation = 'rotate(' + trueHeading + 'deg)';
    var compassStyle = compass.style;
    var northStyle = north.style;

    compassStyle.transform = compassRotation;
    northStyle.transform = northRotation;
}

function onCompassError(error){
    alert("Error with Compass");
}
</script>
</body>
</html>

What just happened?
You implemented a real, cross-platform compass using the PhoneGap API. In the process, you learned how to use a pretty complex feature of mobile device sensors.

An introduction to geolocation
The term geolocation is used in order to refer to the identification process of the real-world geographic location of an object. Devices that are able to detect the user’s position are becoming more common each day and we are now used to getting content based on our location (geo targeting).
Using the **Global Positioning System (GPS)**—a space-based satellite navigation system that provides location and time information consistently across the globe—you can now get the accurate location of a device. During the early 1970s, the US military created Navstar, which is a defense navigation satellite system. Navstar was the system that created the basis for the GPS infrastructure used today by billions of devices. As of October 2014, 68 GPS satellites have been successfully placed in the orbit around the Earth (refer to [http://en.wikipedia.org/wiki/List_of_GPS_satellite_launches](http://en.wikipedia.org/wiki/List_of_GPS_satellite_launches) for a detailed report about the past and planned launches).

The location of a device is represented through a point. This point is comprised of two components: latitude and longitude. There are many methods for modern devices to determine the location information; these include the following:

- GPS
- IP address
- GSM/CDMA cell IDs
- Wi-Fi and Bluetooth MAC address

Each approach delivers the same information; what changes is the accuracy of the device's position. The GPS satellites continuously transmit information that can parse; for example, the general health of the GPS array, roughly where all of the satellites are in orbit, information on the precise orbit or path of the transmitting satellite, and the time of the transmission. The receiver calculates its own position by timing the signals sent by any of the satellites in the array that is visible.

The process of measuring the distance from a point to a group of satellites in order to locate a position is known as **trilateration**. The distance is determined using the speed of light as a constant, along with the time that the signal left the satellites.

The emerging trend in mobile development is GPS-based "people discovery" apps such as Highlight, Sonar, Banjo, and Foursquare. Each app has different features and has been built for different purposes, but all of them share the same killer feature: using location as a piece of metadata in order to filter information according to the user's needs.

**The PhoneGap Geolocation API**

The Geolocation API is not a part of the HTML5 specification, but it is tightly integrated with mobile development. The PhoneGap Geolocation API and the W3C Geolocation API mirror each other; both define the same methods and relative arguments. There are several devices that already implement the W3C Geolocation API; for those devices, you can use native support instead of the PhoneGap API.
As per the HTML specification, the user has to explicitly allow the website or the app to use the device's current position.

The Geolocation API is exposed through the `geolocation` object child of the `navigator` object and consists of the following three methods:

- `getCurrentPosition()`: This returns the device position
- `watchPosition()`: This watches for changes in the device position
- `clearWatch()`: This stops the watcher for the device's position changes

The `watchPosition()` and `clearWatch()` methods work in the same way that the `setInterval()` and `clearInterval()` methods work; in fact, the first one returns an identifier that is passed in to the second one. The `getCurrentPosition()` and `watchPosition()` methods mirror each other and take the same arguments: a success and a failure callback function and an optional configuration object. The configuration object is used in order to specify the maximum age of a cached value of the device's position, in order to set a timeout after which the method will fail and specify whether the application requires only accurate results:

```javascript
var options = {maximumAge: 3000, timeout: 5000, enableHighAccuracy: true};
navigator.geolocation.watchPosition(onSuccess, onFailure, options);
```

Only the first argument is mandatory, but it's recommended to always handle the failure use case.

The success handler function receives a `Position` object as an argument. Accessing its properties, you can read the device's coordinates and the creation timestamp of the object that stores the coordinates:

```javascript
function onSuccess(position) {
    console.log('Coordinates: ' + position.coords);
    console.log('Timestamp: ' + position.timestamp);
}
```

The `coords` property of the `Position` object contains a `Coordinates` object; so far, the most important properties of this object are longitude and latitude. Using those properties, it's possible to start to integrate positioning information as relevant metadata in your app.
The failure handler receives a `PositionError` object as an argument. Using the `code` and the `message` property of this object, you can gracefully handle every possible error:

```javascript
function onError(error) {
  console.log('message: ' + error.message);
  console.log('code: ' + error.code);
}
```

The `message` property returns a detailed description of the error and the `code` property returns an integer; the possible values are represented through the following pseudo constants:

- `PositionError.PERMISSION_DENIED`: This indicates that the user denied the app to use the device's current position
- `PositionError.POSITION_UNAVAILABLE`: This indicates that the position of the device cannot be determined

If you want to recover the last available position when the `POSITION_UNAVAILABLE` error is returned, you have to write a custom plugin that uses the platform-specific API. Android and iOS have this feature. You can find a detailed example at [http://stackoverflow.com/questions/10897081/retrieving-last-known-geolocation-phonegap](http://stackoverflow.com/questions/10897081/retrieving-last-known-geolocation-phonegap).

- `PositionError.TIMEOUT`: This indicates that the specified timeout has elapsed before the implementation could successfully acquire a new `Position` object.

JavaScript doesn't support constants such as Java and other object-oriented programming languages. With the term "pseudo constants," I refer to those values that should never change in a JavaScript app.

One of the most common tasks to perform with the device position information is to show the device location on a map. You can quickly perform this task by integrating Google Maps in your app; the only requirement is a valid API key. To get the key, use the following steps:

1. Visit the APIs' console at [https://code.google.com/apis/console](https://code.google.com/apis/console) and log in with your Google account.
2. Select **APIs & auth** in the left-hand side menu.
3. Select **Google Maps JavaScript API** and activate the service.
Time for action – showing device position with Google Maps

Get ready to add a map renderer to the PhoneGap default app template. Execute the following steps:

1. Open the command-line tool and create a new PhoneGap project named MapSample:
   
   ```sh
   $ cordova create MapSample
   ```

2. Change the working directory to the newly created project:
   
   ```sh
   $ cd MapSample
   ```

3. Add the required platform to the project. For example, we will add Android to this project:
   
   ```sh
   $ cordova platform add android
   ```

4. Add the Geolocation API plugin using the following command:
   
   ```sh
   $ cordova plugin add cordova-plugin-geolocation
   ```

5. Go to the www folder, open the index.html file, remove all the existing content, and add a div element with the id value as #map inside the body tag:
   
   ```html
   <div id='map'></div>
   ```

6. Include the cordova.js file that will be added to the app at runtime:
   
   ```html
   <script type="text/javascript" src="cordova.js"></script>
   ```

7. Add a new script tag to include the Google Maps JavaScript library. Replace the YOUR_API_KEY value with your actual Google Maps API key:
   
   ```html
   <script type="text/javascript"
   src="https://maps.googleapis.com/maps/api/js?key=YOUR_API_KEY&sensor=true">
   </script>
   ```

8. Create a new CSS style to give an appropriate size to the div element and its content:
   
   ```css
   #map{
       width: 280px;
       height: 230px;
       display: block;
       margin: 5px auto;
       position: relative;
   }
   ```
9. In the JavaScript section, define a new function named `initMap`:
   
   ```javascript
   function initMap(lat, long){
       // The code needed to show the map and the
       // device position will be added here
   }
   ```

10. In the body of the function, define an `options` object in order to specify how the map has to be rendered:
    ```javascript
    var options = {
        zoom: 8,
        center: new google.maps.LatLng(lat, long),
        mapTypeId: google.maps.MapTypeId.ROADMAP
    };
    ```

11. Add to the body of the `initMap` function the code to initialize the rendering of the map and to show a marker representing the current device's position over it:
    ```javascript
    var map = new google.maps.Map(document.getElementById('map'), options);
    var markerPoint = new google.maps.LatLng(lat, long);
    var marker = new google.maps.Marker({
        position: markerPoint,
        map: map,
        title: 'Device\'s Location'
    });
    ```

12. Define a function to use as the success handler and call from its body the `initMap` function previously defined:
    ```javascript
    function onSuccess(position){
        var coords = position.coords;
        initMap(coords.latitude, coords.longitude);
    }
    ```
13. Define another function in order to have a failure handler that is able to notify the user that something went wrong:

```javascript
function onFailure(error) {
    alert(error.message);
}
```

14. Go into the deviceready function and add as the last statement the call to the Geolocation API needed to recover the device’s position:

```javascript
navigator.geolocation.getCurrentPosition(app.onSuccess,
app.onFailure, {timeout: 5000, enableAccuracy: false});
```

15. Open the command-line tool, build the app, and then run it on your testing devices:

```
$ cordova build
$ cordova run android
```

Here is the project’s complete code for your reference:

```html
<!DOCTYPE html>
<html>
<head>
<title>GeoLocation Example</title>

<style>
#map{
    width: 280px;
    height: 230px;
    display: block;
    margin: 5px auto;
    position: relative;
}
</style>

</head>

<body>

<div id='map'></div>

<script type="text/javascript" src="cordova.js"></script>
```
<script type="text/javascript" src="https://maps.googleapis.com/maps/api/js?key=YOUR_API_KEY&sensor=true"></script>

<script type="text/javascript">
    var currentHeading = null;

    document.addEventListener("deviceready", onDeviceReady, false);

    function onDeviceReady() {
        navigator.geolocation.getCurrentPosition(onSuccess, onFailure,
        {timeout: 5000, enableAccuracy: false});
    }

    function onSuccess(position){
        var coords = position.coords;
        initMap(coords.latitude, coords.longitude);
    }

    function initMap(lat, long){
        var options = {
            zoom: 8,
            center: new google.maps.LatLng(lat, long),
            mapTypeId: google.maps.MapTypeId.ROADMAP
        };

        var map = new google.maps.Map(document.getElementById('map'),
            options);

        var markerPoint = new google.maps.LatLng(lat, long);

        var marker = new google.maps.Marker({
            position: markerPoint,
            map: map,
            title: 'Device\'s Location'
        });
    }

    function onFailure(error){
        alert(error.message);
    }
</script>
</body>
</html>
Accessing the Device Sensors and Locations API

What just happened?

You integrated Google Maps inside an app. This map is an interactive map most users are familiar with—the most common gestures are already working and the Google Street View controls are already enabled.

To successfully load the Google Maps API on iOS, it’s mandatory to whitelist the googleapis.com and gstatic.com domains. Open the .plist file of the project as source code (right-click on the file and then Open As | Source Code) and add the following array of domains:

```xml
<key>ExternalHosts</key>
<array>
  <string>*.googleapis.com</string>
  <string>*.gstatic.com</string>
</array>
```

Other geolocation data

In the previous example, you only used the latitude and longitude properties of the position object that you received. There are other attributes that can be accessed as properties of the Coordinates object:

- **altitude**: This gives the height of the device, in meters, above the sea level
- **accuracy**: This gives the accuracy level of the latitude and longitude, in meters; it can be used to show a radius of accuracy when mapping the device’s position
- **altitudeAccuracy**: This gives the accuracy of the altitude in meters
- **heading**: This gives the direction of the device in degrees clockwise from true north
- **speed**: This gives the current ground speed of the device in meters per second

The latitude and longitude properties are the best supported of these properties, and the ones that will be most useful when communicating with remote APIs. The other properties are mainly useful if you’re developing an application for which geolocation is a core component of its standard functionality, such as apps that make use of this data to create a flow of information contextualized to the geolocation data. The accuracy property is the most important of these additional features, because as an application developer, you typically won’t know which particular sensor is giving you the location and you can use the accuracy property as a range in your queries to external services.

There are several APIs that allow you to discover interesting data related to a place; among these, the most interesting are the Google Places API and the Foursquare API.
The Google Places and Foursquare online documentation is very well organized and it's the right place to start if you want to dig deeper into these topics. You can access the Google Places docs at https://developers.google.com/maps/documentation/javascript/places and Foursquare at https://developer.foursquare.com/.

Summary

In this chapter, you learned how to work with device sensors to enhance the functionality of your app. You also learned how to get the geolocation information from a device and how to integrate external geolocation service in the app. Furthermore, you continued to gain an understanding of the PhoneGap APIs that allow you to create powerful native apps.

In the next chapter, you will start to work with some advanced concepts of using PhoneGap.
Where to buy this book


Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.