Quick answers to common problems

PhoneGap Mobile Application Development Cookbook

Over 40 recipes to create mobile applications using the PhoneGap API with examples and clear instructions

Foreword by Brian Leroux, Senior Product Manager, PhoneGap Lead and SPACELORD!1!! at Adobe Systems Ltd

Matt Gifford

[PACKT] PUBLISHING open source* community experience distilled

1 Movement and Location: Using the Accelerometer and Geolocation Sensors

In this chapter we will cover the following recipes:

- Detecting device movement using the accelerometer
- Adjusting the accelerometer sensor update interval
- Updating a display object position through accelerometer events
- Obtaining device geolocation sensor information
- Adjusting the geolocation sensor update interval
- Retrieving map data through geolocation coordinates
- Creating a visual compass to show the device direction

Introduction

Mobile devices are incredibly powerful tools that not only allow us to make calls and send messages, but can also help us navigate and find out where we are in the world, thanks to the accelerometer, geolocation, and other sensors.

This chapter will explore how we can access these sensors and make use of this exposed functionality in helpful applications that can be built in any IDE using the PhoneGap API.

Detecting device movement using the accelerometer

The accelerometer captures device motion in the *x*, *y*, and *z* -axis directions. The accelerometer is a motion sensor that detects the change (delta) in movement relative to the current device orientation.

How to do it...

We will use the accelerometer functionality from the PhoneGap API to monitor the feedback from the device:

1. First, create the initial HTML layout and include the required script reference to the `cordova-2.0.0.js` file:

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="user-scalable=no,
            initial-scale=1, maximum-scale=1,
            minimum-scale=1, width=device-width;" />

      <title>Accelerometer Data</title>

      <script type="text/javascript"
               src="cordova-2.0.0.js"></script>

  <!-- Add PhoneGap script here -->

  </head>
  <body>

    <h1>Accelerometer Data</h1>

    <div id="accelerometerData">Obtaining data...</div>

  </body>
</html>
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

2. Below the `Cordova JavaScript` reference, write a new JavaScript tag block and define an event listener to ensure the device is ready and the native code has loaded before continuing:

```
<script type="text/javascript">

  // Set the event listener to run
// when the device is ready
   document.addEventListener(
      "deviceready", onDeviceReady, false);



</script>
```

3. We will now add in the `onDeviceReady` function which will run the `getCurrentAcceleration` method when the native code has fully loaded:

```
// The device is ready so let's
// obtain the current accelerometer data
function onDeviceReady() {
     navigator.accelerometer.getCurrentAcceleration(
      onSuccess, onError);
}
```

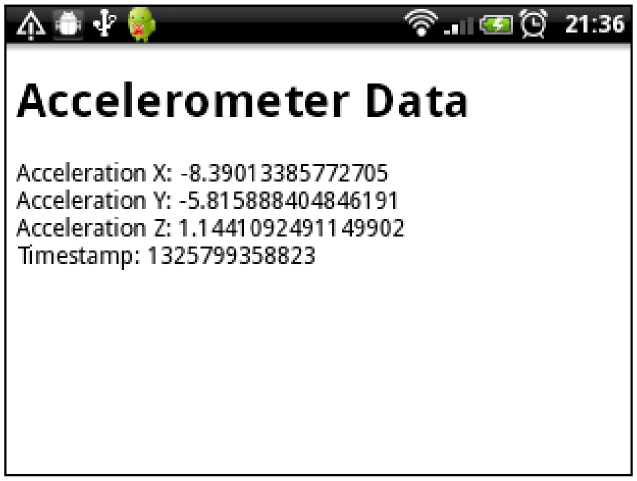
4. Include the `onSuccess` function to handle the returned information from the accelerometer.
5. We now define the `accelerometer div` element to the `accElement` variable to hold our generated accelerometer results.
6. Next, we assign the returned values from the `acceleration` object as the HTML within the `accelerometer div` element for display to the user. The available properties are accessed through the `acceleration` object and applied to the string variable:

```
// Run after successful transaction
// Let's display the accelerometer data
function onSuccess(acceleration) {
  var accElement =
    document.getElementById('accelerometerData');

  accElement.innerHTML  =
    'Acceleration X: ' + acceleration.x + '<br />' +
    'Acceleration Y: ' + acceleration.y + '<br />' +
    'Acceleration Z: ' + acceleration.z + '<br />' +
    'Timestamp: '      + acceleration.timestamp;
}
```

7. Finally, include the `onError` function to deal with any possible issues:

```
// Run if we face an error
// obtaining the accelerometer data
function onError(error) {
  // Handle any errors we may face
  alert('error');
}
```

8. When we run the application on a device, the output will look something like the following screenshot:



Accelerometer Data

Acceleration X: -8.39013385772705
Acceleration Y: -5.815888404846191
Acceleration Z: 1.1441092491149902
Timestamp: 1325799358823

How it works...

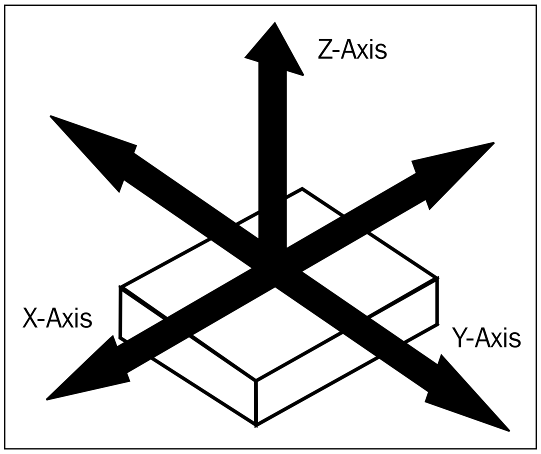
By registering an event listener to the `deviceready` event we are ensuring that the JavaScript code does not run before the native PhoneGap code is executed. Once ready, the application will call the `getCurrentAcceleration` method from the accelerometer API, providing two methods to handle successful transactions and errors respectively.

The `onSuccess` function returns the obtained acceleration information in the form of the following four properties:

- `acceleration.x`: A `number` value, registered in *m/s^2*, that measures the device acceleration across the *X* axis. This is the movement from left to right when the device is placed with the screen facing an upright position. Positive acceleration is obtained as the device is moved to the right, whereas a negative movement is obtained when the device is moved to the left.

- `acceleration.y`: A `Number` value, registered in *m/s*^2, that measures the device acceleration across the *Y* axis. This is the movement from bottom to top when the device is placed with the screen facing an upright position. Positive acceleration is obtained as the device is moved upwards, whereas a negative movement is obtained when the device is moved downwards.
- `acceleration.z`: A `Number` value, registered in *m/s*^2, that measures the device acceleration across the *Z* axis. This is a perpendicular from the face of the device. Positive acceleration is obtained when the device is moved to face towards the sky, whereas a negative movement is obtained when the device is pointed towards the Earth.
- `acceleration.timestamp`: A `DOMTimeStamp` object that measures the amount of milliseconds from the point of the application's initialization. This could be used to store, update, and track changes over a period of time since the last accelerometer update.

The following figure shows the *X*, *Y*, and *Z* Axes in relation to the device:



The `acceleration.x`, `acceleration.y` and `acceleration.z` values returned from the acceleration object previously mentioned include the effect of gravity, which is defined as precisely 9.81 meters per second squared (9.81 m/s^2).

There's more...

Accelerometer data obtained from the device has been used to a great effect in mobile handset games that require balance control and detection of movement including steering, control views, and tilting objects.

> You can check out the official Cordova documentation covering the `getCurrentAcceleration` method and obtaining accelerometer data at:
>
> `http://docs.phonegap.com/en/2.0.0/cordova_accelerometer_accelerometer.md.html#accelerometer.getCurrentAcceleration`.

Adjusting the accelerometer sensor update interval

The `getCurrentAcceleration` method obtains the data from the accelerometer at the time it was called – a single call to obtain a single response object. In this recipe, we'll build an application that allows us to set an interval to obtain a constant update from the accelerometer to detect continual movement from the device.

How to do it...

We will provide additional parameters to a new method available through the PhoneGap API to set the update interval:

1. Firstly, create the initial HTML layout and include the required script reference to the `cordova-2.0.0.js` file:

```
<!DOCTYPE html>
<html>
<head>
       <meta name="viewport" content="user-scalable=no,
            initial-scale=1, maximum-scale=1,
            minimum-scale=1, width=device-width;" />

    <title>Accelerometer Data</title>

      <script type="text/javascript"
           src="cordova-2.0.0.js"></script>

  <!-- Add PhoneGap script here -->

  </head>
  <body>

    <h1>Accelerometer Data</h1>

    <div id="accelerometerData">Obtaining data...</div>

  </body>
</html>
```

2. Below the Cordova JavaScript reference, write a new JavaScript tag block, within which we'll declare a variable called `watchID`.
3. Next, we'll define an event listener to ensure the device is ready and the native code has loaded before continuing:

```
<script type="text/javascript">


  // The watch id variable is set as a
  // reference to the current 'watchAcceleration'
  var watchID = null;


  // Set the event listener to run
// when the device is ready
  document.addEventListener(
      "deviceready", onDeviceReady, false);


</script>
```

4. We will now add in the `onDeviceReady` function which will run a method called `startWatch` once the native code has fully loaded:

```
// The device is ready so let's
// start watching the acceleration
function onDeviceReady() {
  startWatch();
}
```

5. We'll now write the `startWatch` function. Firstly, we'll create a variable called `options` to hold the optional `frequency` parameter, set to 3000 milliseconds (three seconds).
6. We will then set the initial `disabled` properties of two buttons that will allow the user to start and stop the acceleration detection.
7. Next we will assign the `watchAcceleration` to the previously defined `watchID` variable. This will allow us to check for a value or if it is still set to `null`.

8. As well as defining the success and error function names we are also sending the `options` variable into the method call, which contains the `frequency` value:

```
// Watch the acceleration at regular
// intervals as set by the frequency
function startWatch() {

  // Set the frequency of updates
  // from the acceleration
  var options = { frequency: 3000 };

  // Set attributes for control buttons
  document.getElementById('startBtn').disabled = true;
  document.getElementById('stopBtn').disabled = false;

  // Assign watchAcceleration to the watchID variable
  // and pass through the options array
  watchID = navigator.accelerometer.watchAcceleration(
            onSuccess, onError, options);
}
```

9. With the `startWatch` function written, we now need to provide a method to stop the detection of the acceleration. This firstly checks the value of the `watchID` variable. If this is not `null` it will stop watching the acceleration using the `clearWatch` method, passing in the `watchID` parameter before resetting this variable back to `null`.
10. We then reference the `accelerometer div` element and set its value to a user-friendly message.
11. Next, we reassign the `disabled` properties for both of the control buttons to allow the user to start watching again:

```
// Stop watching the acceleration
function stopWatch() {

  if (watchID) {
    navigator.accelerometer.clearWatch(watchID);
      watchID = null;

    var element =
      document.getElementById('accelerometerData');

    element.innerHTML =
      'No longer watching your acceleration.'

    // Set attributes for control buttons
    document.getElementById('startBtn').disabled = false;
```

```
        document.getElementById('stopBtn').disabled = true;

      }

    }
```

12. Now we need to create the `onSuccess` method, which will be run after a successful update response. We assign the returned values from the `acceleration` object as the HTML within the `accelerometer div` element for display to the user. The available properties are accessed through the `acceleration` object and applied to the string variable:

```
// Run after successful transaction
// Let's display the accelerometer data
function onSuccess(acceleration) {
  var element = document.getElementById('accelerometerData');
  element.innerHTML =
    'Acceleration X: ' + acceleration.x + '<br />' +
    'Acceleration Y: ' + acceleration.y + '<br />' +
      'Acceleration Z: ' + acceleration.z + '<br />' +
      'Timestamp: '      + acceleration.timestamp + '<br />';
}
```

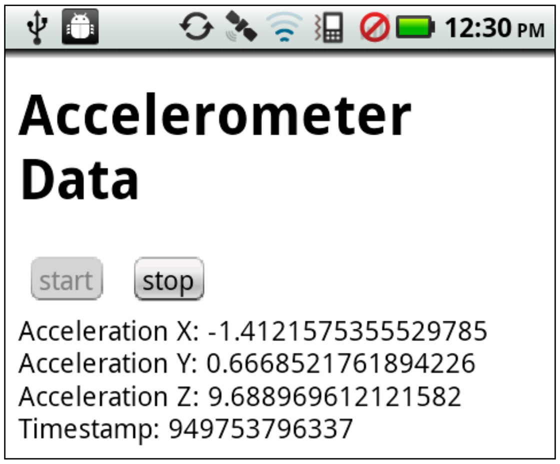
13. We also need to supply the `onError` method to catch any possible issues with the request. Here we will output a user-friendly message, setting it as the value of the `accelerometerData div` element:

```
// Run if we face an error
// obtaining the accelerometer data
function onError() {
  // Handle any errors we may face
  var element = document.getElementById('accelerometerData');
    element.innerHTML =
    'Sorry, I was unable to access the acceleration data.';
}
```

14. Finally, we will add in the two button elements, both of which will have an `onClick` attribute set to either start or stop watching the device acceleration:

```
<body>
    <h1>Accelerometer Data</h1>

  <button id="startBtn"
        onclick="startWatch()">start</button>

  <button id="stopBtn"
        onclick="stopWatch()">stop</button>
        <div id="accelerometerData">Obtaining data...</div>

    </body>
```

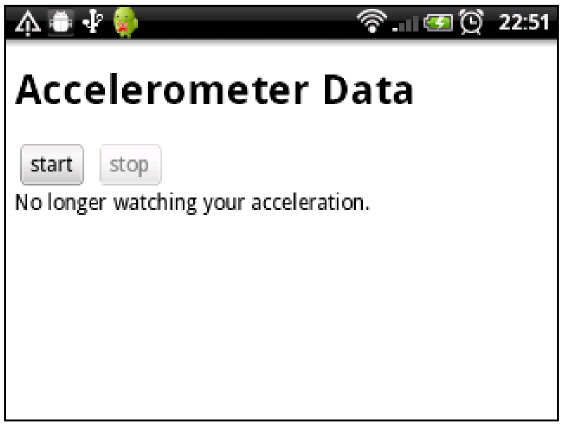
15. The results will appear similar to the following screenshot:



Accelerometer Data

Acceleration X: -1.4121575355529785
Acceleration Y: 0.6668521761894226
Acceleration Z: 9.688969612121582
Timestamp: 949753796337

16. Stopping the acceleration watch will look something like the following screenshot:



Accelerometer Data

No longer watching your acceleration.

How it works...

By registering an event listener to the `deviceready` event we are ensuring that the JavaScript code does not run before the native PhoneGap code is executed. Once ready, the application will call the `startWatch` function, within which the desired frequency interval for the acceleration updates is set.

The `watchAcceleration` method from the PhoneGap API retrieves the device's current acceleration data at the interval specified. If the interval is not passed through, it defaults to 10000 milliseconds (ten seconds). Each time an update has been obtained, the `onSuccess` method is run to handle the data as you wish, in this case displaying the results on the screen.

The `watchID` variable contains a reference to the watch interval and is used to stop the watching process by being passed in to the `clearWatch` method from the PhoneGap API.

There's more...

In this example the `frequency` value for the accelerometer update interval was set at 3000 milliseconds (three seconds). Consider writing a variation on this application that allows the user to manually change the interval value using a slider or by setting the desired value into an input box.

> You can find out more about the `watchAcceleration` method via the official Cordova documentation: `http://docs.phonegap.com/en/2.0.0/cordova_accelerometer_accelerometer.md.html#accelerometer.watchAcceleration`.

Updating a display object position through accelerometer events

Developers can make use of the accelerometer sensor and continual updates provided by it for many things including motion-detection games as well as updating the position of an object on the screen.

How to do it...

We will use the device's accelerometer sensor on continual update to move an element around the screen as a response to device movement. This is achieved through the following steps:

1. Let's start by creating our initial HTML layout. Include the Cordova JavaScript reference in the `head` tag to import the required library.
2. Within the `body` tag create two `div` elements. Set the first with the `id` attribute equal to dot. This will be the element we move around the screen of the device.
3. The second `div` element will have the ID of `accelerometerData` and will be the container into which our returned acceleration data will be output:

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

```
        <meta name="viewport" content="user-scalable=no,
              initial-scale=1, maximum-scale=1,
              minimum-scale=1, width=device-width;" />

      <title>Accelerometer Movement</title>
      <script type="text/javascript"
           src="cordova-2.0.0.js"></script>

    </head>
    <body>
      <h1>Accelerometer Movement</h1>

      <div id="dot"></div>

      <div id="accelerometerData">Obtaining data...</div>

    </body>
  </html>
```

4. We can now start with our custom scripting and PhoneGap implementation. Add a `script` tag block before the closing `head` tag to house our code:

```
<head>
    <meta name="viewport" content="user-scalable=no,
            initial-scale=1, maximum-scale=1,
            minimum-scale=1, width=device-width;" />

    <title>Accelerometer Movement</title>
  <script type="text/javascript"
        src="cordova-2.0.0.js"></script>

  <script type="text/javascript">

  </script>

</head>
```

5. Before we dive into the core code, we need to declare some variables. Here we are setting a default value for `watchID` as well as the radius for the circle display object we will be moving around the screen:

```
// The watch id variable is set as a
// reference to the current `watchAcceleration`
var watchID = null;

// The radius for our circle object
var radius   = 50;
```

6. We now need to declare the event listener for PhoneGap, as well as the `onDeviceReady` function, which will run once the native PhoneGap code has been loaded:

```
// Set the event listener to run when the device is ready
document.addEventListener("deviceready",
        onDeviceReady, false);

// The device is ready so let's
// start watching the acceleration
function onDeviceReady() {

  startWatch();

}
```

7. The `onDeviceReady` function will execute the `startWatch` method, which sets required the `frequency` value for accelerometer updates and makes the request to the device to obtain the information:

```
// Watch the acceleration at regular
// intervals as set by the frequency
function startWatch() {

  // Set the frequency of updates from the acceleration
   var options = { frequency: 100 };

  // Assign watchAcceleration to the watchID variable
  // and pass through the options array
  watchID =
    navigator.accelerometer.watchAcceleration(
      onSuccess, onError, options);
}
```

8. With the request made to the device we now need to create the success and error handling methods. The `onSuccess` function is first, and this will deal with the movement of our object around the screen.
9. To begin with we need to declare some variables that manage the positioning of our element on the device:

```
function onSuccess(acceleration) {

  // Initial X Y positions
  var x = 0;
  var y = 0;
```

```
    // Velocity / Speed
    var vx = 0;
    var vy = 0;

    // Acceleration
    var accelX = 0;
    var accelY = 0;

    // Multiplier to create proper pixel measurements
    var vMultiplier   =      100;

    // Create a reference to our div elements
    var dot = document.getElementById('dot');
    var accelElement =
          document.getElementById('accelerometerData');


    // The rest of the code will go here

  }
```

10. The returned `acceleration` object contains the information we need regarding the position on the *x* and *y* axes of the device. We can now set the acceleration values for these two axis into our variables and work out the velocity for movement.
11. To correctly interpret the acceleration results into pixels we can use the `vMultiplier` variable to convert the `x` and `y` into pixels:

```
accelX = acceleration.x;
accelY = acceleration.y;

vy = vy + -(accelY);
vx = vx + accelX;

y = parseInt(y + vy * vMultiplier);
x = parseInt(x + vx * vMultiplier);
```

12. We need to ensure that our display object doesn't move out of sight and to keep it within the bounds of the screen:

```
if (x<0) { x = 0; vx = 0; }
if (y<0) { y = 0; vy = 0; }

if (x>document.documentElement.clientWidth-radius) {
  x = document.documentElement.clientWidth-radius; vx = 0;
}
```

```
if (y>document.documentElement.clientHeight-radius) {
  y = document.documentElement.clientHeight-radius; vy = 0;
}
```

13. Now that we have the correct `x` and `y` coordinates we can apply them to the style of the `dot` element position. Let's also create a string message containing the properties returned from the `acceleration` object as well as the display coordinates that we have created:

```
// Apply the position to the dot element
dot.style.top  = y + "px";
dot.style.left = x + "px";

// Output the acceleration results to the screen
accelElement.innerHTML =
  'Acceleration X: '      + acceleration.x + '<br />' +
  'Acceleration Y: '      + acceleration.y + '<br />' +
  'Acceleration Z: '      + acceleration.z + '<br />' +
  'Timestamp: '           + acceleration.timestamp + '<br />' +
  'Move Top: '                   + y + 'px<br />' +
  'Move Left: '                  + x + 'px';
```

14. Our call to the accelerometer also requires the error handler, so let's write that now. We'll create a simple string message and insert it into the `div` element to inform the user that we encountered a problem:

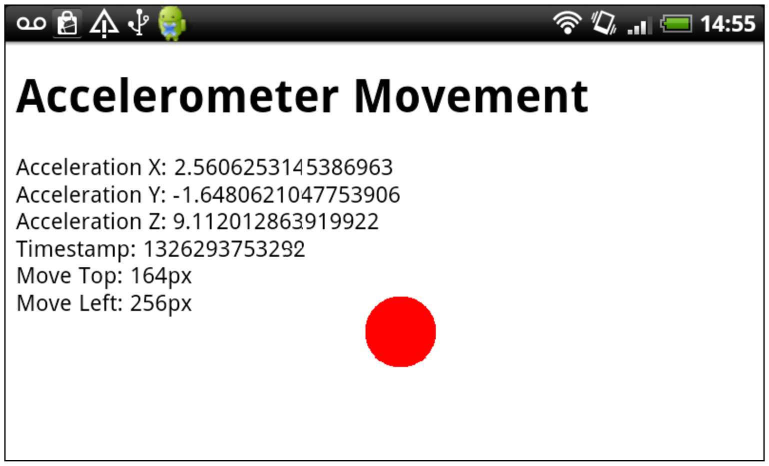
```
// Run if we face an error
// obtaining the accelerometer data
function onError() {

  // Handle any errors we may face
  var accelElement =
      document.getElementById('accelerometerData');

  accelElement.innerHTML =
    'Sorry, I was unable to access the acceleration data.';
}
```

15. Finally, we'll add in some CSS to create the dot marker used to display the position on our device:

```
<style>
div#dot {
  border-radius: 14px;
  width: 25px;
  height: 25px;
  background: #ff0000;
```

```
      position: absolute;
      top: 0px;
      left: 0px;
    }
    </style>
```

16. When we run the application, we can move the element around the screen by titling the device. This would look something like the following screenshot:



Accelerometer Movement

```
Acceleration X: 2.5606253145386963
Acceleration Y: -1.6480621047753906
Acceleration Z: 9.112012863919922
Timestamp: 1326293753292
Move Top: 164px
Move Left: 256px
```

How it works...

By implementing a constant request to watch the acceleration and retrieve movement results from the device, we can pick up changes from the accelerometer sensor. Through some simple JavaScript we can respond to these changes and update the position of an element around the screen based upon the returned sensor information.

In this recipe we are easily changing the position of the `dot` element by calculating the correct *X* and *Y* axes to place it on the screen. We are also taking extra care to ensure that the element stays within the bounds of the screen by using some conditional statements to check the current position, the radius of the element, and the dimensions of the screen itself.

See also

- The *Detecting device movement using the accelerometer* recipe

Obtaining device geolocation sensor information

Geolocation and the use of **Global Positioning Satellites** (**GPS**) allow developers to create dynamic real-time mapping, positioning, and tracking applications. Using the available geolocation methods we can retrieve a detailed set of information and properties to create location-aware applications. We can obtain the user's location if they are connected via the mobile data network or Wi-Fi.

How to do it...

We will use the geolocation functionality from the PhoneGap API to monitor the feedback from the device and obtain the relevant location information:

1. Firstly, create the initial HTML layout and include the required script reference to the `cordova-2.0.0.js` file:

```
<!DOCTYPE html>
<html>
<head>
      <meta name="viewport" content="user-scalable=no,
            initial-scale=1, maximum-scale=1,
            minimum-scale=1, width=device-width;" />

    <title>Geolocation Data</title>

      <script type="text/javascript"
           src="cordova-2.0.0.js"></script>

  <!-- Add PhoneGap script here -->

  </head>
  <body>

    <h1>Geolocation Data</h1>

    <div id="geolocationData">Obtaining data...</div>

  </body>
</html>
```

2. Write a new JavaScript tag block beneath the Cordova JavaScript reference. Within this let's define an event listener to ensure the device is ready:

```
<script type="text/javascript" charset="utf-8">


// Set the event listener to run when the device is ready
document.addEventListener(
        "deviceready", onDeviceReady, false);

</script>
```

3. Let's now add the `onDeviceReady` function. This will execute the `geolocation.getCurrentPosition` method from the PhoneGap API once the native code has fully loaded:

```
// The device is ready so let's
// obtain the current geolocation data
function onDeviceReady() {
  navigator.geolocation.getCurrentPosition(
        onSuccess, onError);
}
```

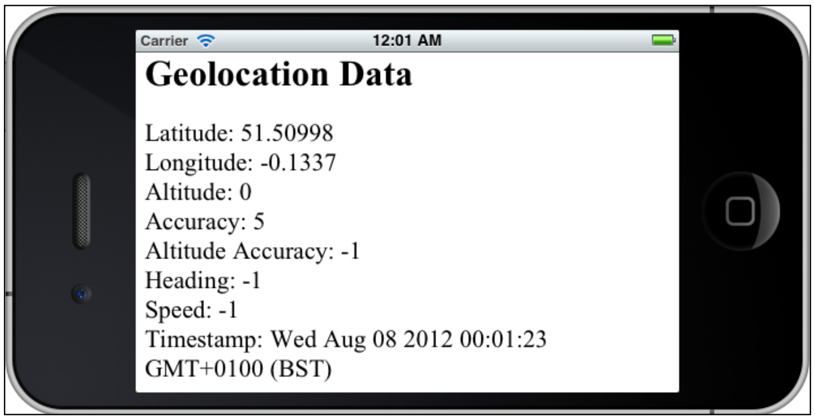
4. Include the `onSuccess` function to handle the returned `position` object from the geolocation request.
5. Let's then create a reference to the `geolocationData div` element and assign it to the variable `geoElement`, which will hold our generated position results.
6. Next we can assign the returned values as a formatted string, which we'll set as the HTML content within the `geolocationData div` element. The available properties are accessed through the `position` object:

```
// Run after successful transaction
// Let's display the position data
function onSuccess(position) {

  var geoElement =
        document.getElementById('geolocationData');

  geoElement .innerHTML =
    'Latitude: '  + position.coords.latitude + '<br />' +
      'Longitude: ' + position.coords.longitude + '<br />' +
      'Altitude: '  + position.coords.altitude + '<br />' +
      'Accuracy: '  + position.coords.accuracy + '<br />' +
      'Altitude Accuracy: ' +
        position.coords.altitudeAccuracy + '<br />' +
      'Heading: ' + position.coords.heading  + '<br />' +
      'Speed: '   + position.coords.speed + '<br />' +
      'Timestamp: ' + position.timestamp + '<br />';

}
```

7. Finally, let's include the `onError` function to handle any possible errors that may arise.
8. Depending on the existence of an error, we will use the value of the returned error code to determine which message to display to the user. This will be set as the HTML content of the `geolocationData` `div` element:

```
// Run if we face an error
// obtaining the position data
function onError(error) {

  var errString = '';

  // Check to see if we have received an error code
  if(error.code) {

    // If we have, handle it by case
    switch(error.code)
    {
      case 1: // PERMISSION_DENIED
      errString =
          'Unable to obtain the location information ' +
          'because the device does not have permission '+
          'to the use that service.';
      break;
      case 2: // POSITION_UNAVAILABLE
        errString =
          'Unable to obtain the location information ' +
          'because the device location could not ' +
          'be determined.';
      break;
      case 3: // TIMEOUT
        errString =
          'Unable to obtain the location within the ' +
          'specified time allocation.';
      break;
      default: // UNKOWN_ERROR
        errString =
          'Unable to obtain the location of the ' +
          'device due to an unknown error.';
      break;
    }

  }

  // Handle any errors we may face
  var element = document.getElementById('geolocationData');
   element.innerHTML = errString;

}
```

9. When we run the application on a device, the output will look something like the following screenshot:

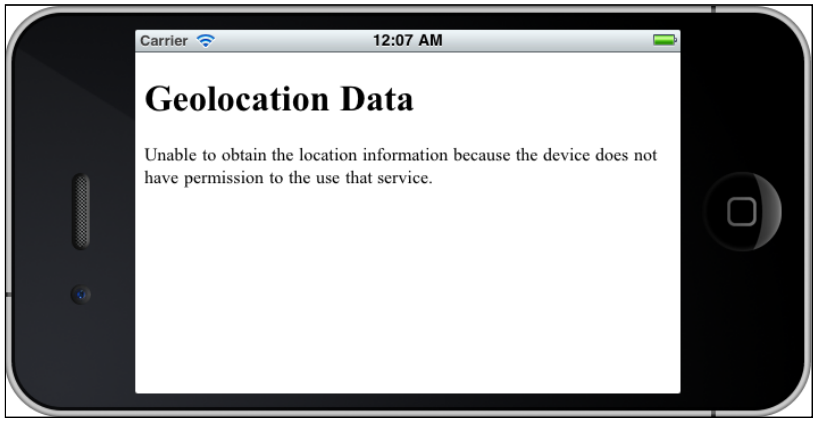


Carrier 12:01 AM

Geolocation Data

Latitude: 51.50998
Longitude: -0.1337
Altitude: 0
Accuracy: 5
Altitude Accuracy: -1
Heading: -1
Speed: -1
Timestamp: Wed Aug 08 2012 00:01:23 GMT+0100 (BST)

10. If we face any errors, the resulting output will look something like the following screenshot:



Carrier 12:07 AM

Geolocation Data

Unable to obtain the location information because the device does not have permission to the use that service.

How it works...

As soon as the device is ready and the native PhoneGap code has been initiated on the device, the application will execute the `getCurrentPosition` method from the `geolocation` API. We have defined an `onSuccess` method to manage the output and handling of a successful response, and have also specified an `onError` method to catch any errors and act accordingly.

The `onSuccess` method returns the obtained geolocation information in the form of the `position` object, which contains the following properties:

- `position.coords`: A `Coordinates` object that holds the geographic information returned from the request. This object contains the following properties:
    - `latitude`: A `Number` value ranging from -90.00 to +90.00 that specifies the latitude estimate in decimal degrees.
    - `longitude`: A `Number` value ranging from -180.00 to +180.00 that specifies the longitude estimate in decimal degrees.
    - `altitude`: A `Number` value that specifies the altitude estimate in meters above the **World Geodetic System** (**WGS**) 84 ellipsoid. Optional.
    - `accuracy`: A `Number` value that specifies the accuracy of the latitude and longitude accuracy in meters. Optional.
    - `altitudeAccuracy`: A `Number` value that specifies the accuracy of the altitude estimate in meters. Optional.
    - `heading`: A `Number` value that specifies the current direction of movement in degrees, counting clockwise in relation to true north. Optional.
    - `speed`: A `Number` value that specifies the current ground speed of the device in meters per second. Optional.
- `position.timestamp`: A `DOMTimeStamp` object that signifies the time that the geolocation information was received and the `Position` object was created.

  The properties available within the `position` object are quite comprehensive and detailed.

For those marked as 'optional', the value will be set and returned as `null` if the device cannot provide a value.

The `onError` method returns a `PositionError` object if an error is detected during the request. This object contains the following two properties:

- `code`: A `Number` value that contains a numeric code for the error.
- `message`: A `String` object that contains a human-readable description of the error.

The errors could relate to insufficient permissions needed to access the geolocation sensors on the device, the inability to locate the device due to issues with obtaining the necessary GPS information, a timeout on the request, or the occurrence of an unknown error.

There's more...

The exposed geolocation API accessible through PhoneGap is based on the *W3C Geolocation API Specification*. Many modern browsers and devices already have this functionality enabled. If any device your application runs on already implements this specification, it will use the built-in support for the API and not PhoneGap's implementation.

> You can find out more about Geolocation and the `getCurrentPosition` method via the official Cordova documentation at: `http://docs.phonegap.com/en/2.0.0/cordova_geolocation_geolocation.md.html#geolocation.getCurrentPosition`.

Adjusting the geolocation sensor update interval

Through the use of the `getCurrentPosition` method, we can retrieve a single reference to the device location using GPS coordinates. In this recipe, we'll create the functionality to obtain the current location based on a numeric interval to receive constant updated information.

How to do it...

We are able to pass through an optional parameter containing various arguments to set up an interval and improve accuracy:

1. Create the HTML layout for the application, including the required `cordova-2.0.0.js` file:

```
<!DOCTYPE html>
<html>
<head>
       <meta name="viewport" content="user-scalable=no,
            initial-scale=1, maximum-scale=1,
            minimum-scale=1, width=device-width;" />
    <title>Geolocation Data</title>

      <script type="text/javascript"
           src="cordova-2.0.0.js"></script>

  <!-- Add PhoneGap script here -->

  </head>
  <body>
    <h1>Geolocation Data</h1>

    <div id="geolocationData">Obtaining data...</div>

  </body>
</html>
```

2. Below the Cordova JavaScript reference, write a new JavaScript tag block. Within this we'll declare a new variable called `watchID`.
3. Next, we'll write the event listener to continue once the device is ready:

```
<script type="text/javascript">

  // The watch id variable is set as a
  // reference to the current 'watchPosition'
  var watchID = null;

  // Set the event listener to run
// when the device is ready
  document.addEventListener(
      "deviceready", onDeviceReady, false);


</script>
```

4. Let's now add the `onDeviceReady` function which will execute a method called `startWatch`, written as follows:

```
// The device is ready so let's
// start watching the position
function onDeviceReady() {

  startWatch();

}
```

5. We can now create the `startWatch` function. Firstly, let's create the `options` variable to hold the optional parameters we can pass through to the method. Set the `frequency` value to 5000 milliseconds (five seconds) and set `enableHighAccuracy` to true.
6. Next we will assign the `watchPosition` method to the previously defined variable `watchID`. We'll use this variable to check if the location is currently being watched.

7. To pass through the extra parameters we have set, we send the `options` variable into the `watchPosition` method:

```
function startWatch() {

  // Create the options to send through
  var options = {
      enableHighAccuracy: true
  };

  // Watch the position and update
  // when a change has been detected
  watchID =
    navigator.geolocation.watchPosition(
      onSuccess, onError, options);

}
```

8. With the initial call methods created, we can now write the `onSuccess` function, which is executed after a successful response. The `position` object from the response is sent through as an argument to the function.
9. Declare some variables to store detailed information obtained from the response in the form of the `timestamp`, `latitude`, `longitude`, and `accuracy` variables. We'll also create the element variable to reference the `geolocationData div` element, within which our information will be displayed.
10. The returned information is then assigned to the relevant variables by accessing the properties from the `position` object.
11. Finally, we apply the populated variables to a concatenated string which we'll set as the HTML within the `div` element:

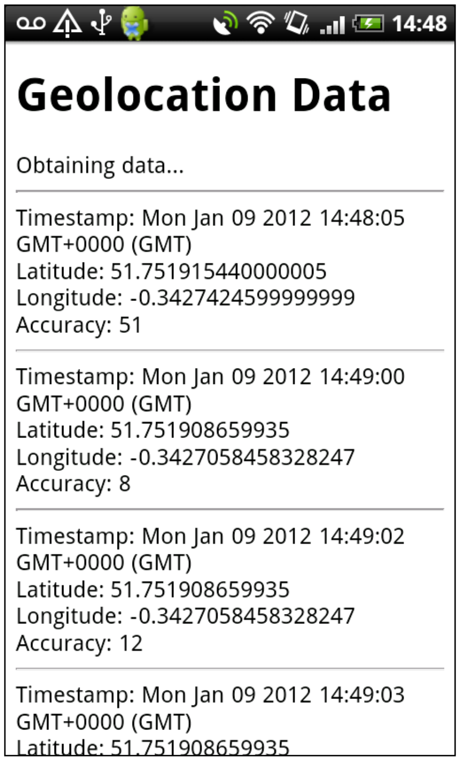
```
// Run after successful transaction
// Let's display the position data
function onSuccess(position) {

  var timestamp, latitude, longitude, accuracy;

  var element = document.getElementById('geolocationData');

  timestamp = new Date(position.timestamp);
  latitude = position.coords.latitude;
  longitude = position.coords.longitude;
  accuracy     = position.coords.accuracy;

  element.innerHTML +=
      '<hr />' +
      'Timestamp: ' + timestamp + '<br />' +
      'Latitude: ' + latitude    + '<br />' +
      'Longitude: ' + longitude + '<br />' +
      'Accuracy: ' + accuracy    + '<br />';
}
```

12. With the `onSuccess` method created, let's now write the `onError` function to handle any errors that we may face following the response:

```
// Run if we face an error
// obtaining the position data
function onError(error) {

  var errString = '';

  // Check to see if we have received an error code
  if(error.code) {
    // If we have, handle it by case
    switch(error.code)
    {
      case 1: // PERMISSION_DENIED
        errString =
          'Unable to obtain the location information ' +
          'because the device does not have permission '+
          'to the use that service.';
      break;
      case 2: // POSITION_UNAVAILABLE
        errString =
          'Unable to obtain the location information ' +
          'because the device location could not be ' +
          'determined.';
      break;
      case 3: // TIMEOUT
        errString =
          'Unable to obtain the location within the ' +
          'specified time allocation.';
      break;
      default: // UNKOWN_ERROR
        errString =
          'Unable to obtain the location of the ' +
          'device to an unknown error.';
      break;
    }

  }
```

```
        // Handle any errors we may face
        var element = document.getElementById('geolocationData');
         element.innerHTML = errString;

      }
```

13. When we run the application, the output will be similar to the following screenshot:



14:48

Geolocation Data

```
Obtaining data...

Timestamp: Mon Jan 09 2012 14:48:05
GMT+0000 (GMT)
Latitude: 51.751915440000005
Longitude: -0.3427424599999999
Accuracy: 51

Timestamp: Mon Jan 09 2012 14:49:00
GMT+0000 (GMT)
Latitude: 51.751908659935
Longitude: -0.3427058458328247
Accuracy: 8

Timestamp: Mon Jan 09 2012 14:49:02
GMT+0000 (GMT)
Latitude: 51.751908659935
Longitude: -0.3427058458328247
Accuracy: 12

Timestamp: Mon Jan 09 2012 14:49:03
GMT+0000 (GMT)
Latitude: 51.751908659935
```

How it works...

The `watchPosition` method from the PhoneGap API runs as an asynchronous function, constantly checking for changes to the device's current position. Once a change in position has been detected, it will return the current geographic location information in the form of the `position` object.

With every successful request made on the continuous cycle, the `onSuccess` method is executed and formats the data for output onto the screen.

There's more...

There are three optional parameters that can be sent into either the `getCurrentPosition` or `watchPosition` method. They are as follows:

- `enableHighAccuracy`: A `Boolean` value that specifies whether or not you would like to obtain the best possible location results from the request. By default (false), the position will be retrieved using the mobile or cell network. If set to true, more accurate methods will be used to locate the device, for example, using satellite positioning.
- `timeout`: A `Number` value that defines the maximum length of time in milliseconds to obtain the successful response.
- `maximumAge`: A `Number` value that defines if a cached position younger than the specified time in milliseconds can be used.

[Android devices will not return a successful geolocation result unless `enableHighAccuracy` is set to `true`]

Clearing the interval

The continual location requests can be stopped by the user or through the application using an interval timer by employing the use of the `clearWatch` method, available within the PhoneGap API. The method to clear the interval and stop watching location data is identical to the method used when clearing accelerometer data obtained from continual updates.

See also

- The *Adjusting the accelerometer sensor update interval* recipe

Retrieving map data through geolocation coordinates

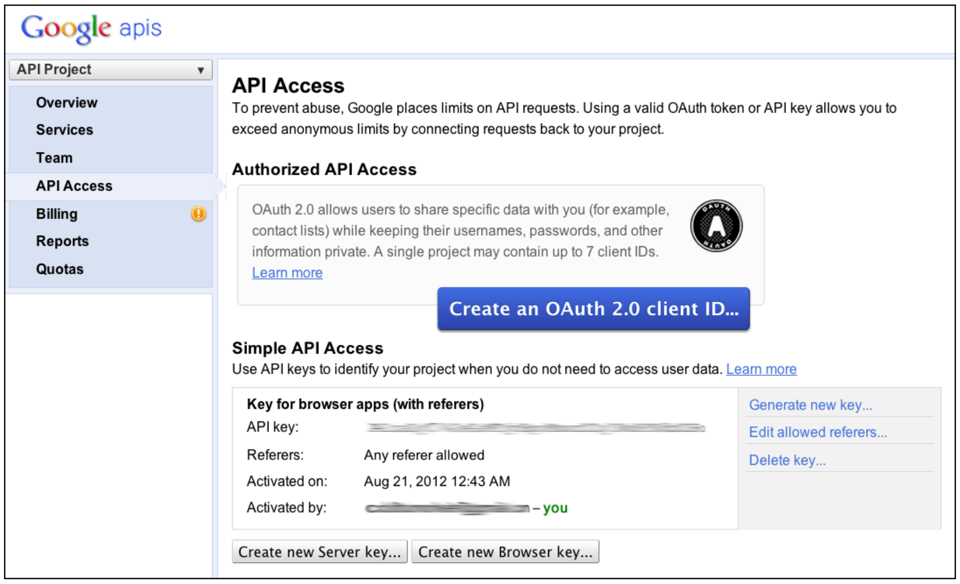
In this recipe, we will examine how to render a map on the screen and generate a marker based on latitude and longitude coordinates reported by the device geolocation sensors using the Google Maps API for JavaScript.

Getting ready

Before we can continue with coding the application in this recipe, we must first prepare the project and obtain access to the Google Maps services:

1. Firstly we need to sign up for a **Google Maps API key**. Visit `https://code.google.com/apis/console/` and log in with your Google account.

2. Select **Services** from the left-hand side menu and activate the **Google Maps API v3 service**.
3. Once the service has been activated you will be presented with your API key, available from the **API Access** page. You will find the key displayed in the **Simple API Access** section of the page, as shown in the following screenshot:



API Access

To prevent abuse, Google places limits on API requests. Using a valid OAuth token or API key allows you to exceed anonymous limits by connecting requests back to your project.

Authorized API Access

OAuth 2.0 allows users to share specific data with you (for example, contact lists) while keeping their usernames, passwords, and other information private. A single project may contain up to 7 client IDs. Learn more

Simple API Access

Use API keys to identify your project when you do not need to access user data. Learn more

Key for browser apps (with referers)

API key: [redacted]

Referers: Any referer allowed

Activated on: Aug 21, 2012 12:43 AM

Activated by: [redacted] – you

4. We can now proceed with the recipe.

You can find out more about the Google Maps API from the official documentation: `https://developers.google.com/maps/documentation/javascript/`.

How to do it...

We'll use the device's GPS ability to obtain the geolocation coordinates, build and initialize the map canvas, and display the marker for our current position:

1. Create the basic HTML layout for our page:

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="user-scalable=no,
            initial-scale=1, maximum-scale=1,
            minimum-scale=1, width=device-width;" />
    <title>You are here...</title>
  </head>
  <body>


  </body>
</html>
```

2. Let's include the required JavaScript for the Google Maps API within the `head` tag. Append your API key into the query string in the script `src` attribute.
3. Next, add the `cordova-2.0.0.js` reference and create another JavaScript tag block to hold our custom code:

```
<head>
  <meta name="viewport" content="user-scalable=no,
            initial-scale=1, maximum-scale=1,
            minimum-scale=1, width=device-width;" />

  <title>You are here...</title>

  <script type="text/javascript"
  src="http://maps.googleapis.com/maps/api/js?key=your_api_
key&sensor=true">
    </script>

  <script type="text/javascript"
        src="cordova-2.0.0.js"></script>

  <script type="text/javascript">

  // Custom PhoneGap code goes here

  </script>

</head>
```

When we included the Google Maps API JavaScript into our document, we set the sensor query parameter to true. If we were only allowing the user to manually input coordinates without automatic detection this could have been set to false. However, we are using the data obtained from the device's sensor to automatically retrieve our location.

4. Let's start creating our custom code within the JavaScript tag block. First, we'll create the event listener to ensure the device is ready and we'll also create the `onDeviceReady` method, which will run using the listener:

   ```
   // Set the event listener to run when the device is ready
   document.addEventListener(
           "deviceready", onDeviceReady, false);

   // The device is ready, so let's
   // obtain the current geolocation data
   function onDeviceReady() {
     navigator.geolocation.getCurrentPosition(
       onSuccess,
       onError
     );
   }
   ```

5. Next we can write the `onSuccess` method, which will give us access to the returned location data via the `position` object.
6. Let's take the `latitude` and `longitude` information obtained from the device geolocation sensor response and create a `latLng` object which we will send into the `Map` object when we initialize the component.
7. We will then set the options for our `Map`, setting the center of it to the coordinates we set into the `latLng` variable. Not all of the Google Map controls translate well to the small screen, especially in terms of usability. We can define which controls we would like to use. In this case we'll accept the `zoomControl` but not the `panControl`.
8. To define the `Map` object itself we reference a `div` element and pass through the `mapOptions` variable we have previously declared.
9. To close off this method, let's now create a `Marker` variable to display at the exact location
   as set in the `latLng` variable:

   ```
   // Run after successful transaction
   // Let's display the position data
   function onSuccess(position) {

     var latLng =
         new google.maps.LatLng(
             position.coords.latitude,
               position.coords.longitude);

     var mapOptions = {
           center: latLng,
           panControl: false,
   ```

```
          zoomControl: true,
          zoom: 16,
          mapTypeId: google.maps.MapTypeId.ROADMAP
        };

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

    var marker = new google.maps.Marker({
          position: latLng,
          map: map
        });

  }
```

10. To ensure we correctly handle any errors that may occur, let's now include the `onError` function which will display the specific string message according to the error within a `div` element:

```
// Run if we face an error
// obtaining the position data
function onError(error) {

  var errString = '';

  // Check to see if we have received an error code
  if(error.code) {
    // If we have, handle it by case
    switch(error.code)
    {
      case 1: // PERMISSION_DENIED
        errString =
          'Unable to obtain the location information ' +
          'because the device does not have permission '+
          'to the use that service.';
      break;
      case 2: // POSITION_UNAVAILABLE
        errString =
          'Unable to obtain the location information ' +
          'because the device location could not be ' +
          'determined.';
```

```
            break;
            case 3: // TIMEOUT
              errString =
                'Unable to obtain the location within the ' +
                'specified time allocation.';
            break;
            default: // UNKOWN_ERROR
              errString =
                'Unable to obtain the location of the ' +
                'device due to an unknown error.';
            break;
          }

        }

        // Handle any errors we may face
        var element = document.getElementById('map_holder');
        element.innerHTML = errString;
      }
```

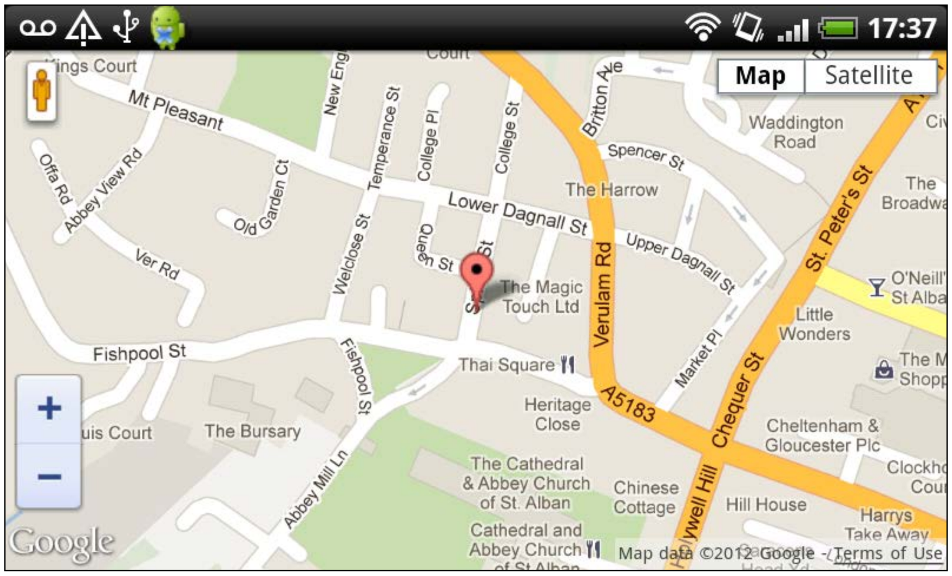
11. With the `body` tag, let's include the `div` element into which the map will be displayed:

```
<body>

    <div id="map_holder"></div>

</body>
```

12. Finally, add a `style` block within the `head` tag to supply some essential formatting to the page and the `map` element:

```
<style type="text/css">
  html { height: 100% }
   body { height: 100%; margin: 0; padding: 0 }
  #map_holder { height: 100%; width: 100%; }
</style>
```

13. Running the application on the device, the result will look similar to this:



How it works...

Thanks to the use of exposed mapping services such as Google Maps we are able to perform geolocation updates from the device and use the obtained data to create rich, interactive visual mapping applications.

In this example, we centered the `Map` using the device coordinates and also created a `Marker` overlay to place upon the mark for easy visual reference.

The available APIs for mapping services such as this are incredibly detailed and contain many functions and methods to assist you in creating your location-based tools and applications. Some services also set limits on the amount of requests made to the API, so make sure you are aware of any restrictions in place.

There's more...

We used the Google Maps API for JavaScript in this recipe. There are variations on the API level offered by Google, and other mapping systems are also available through other providers such as MapQuest, MultiMap, and Yahoo! Maps. Explore the alternatives and experiment to see if a particular solution suits your application better than the others.

Static maps

In this recipe, we used the dynamic Google Map API. We did this so that we could use the zoom controls and provide our user with a certain level of interaction by being able to drag the map.

As an alternative, you could use the Google Static Map service, which simplifies the code needed to generate a map, and will return a static image showing the location.

You can choose to use an API key with this service, but it is not required. You will still have to enable the API in the same way we enabled the API Access at the start of this recipe.

Consider the following code, which is an amendment to the `onSuccess` method, which runs after the geolocation data has been obtained:

```
// Run after successful transaction
// Let's display the position data
function onSuccess(position) {
  var mapOutput = '<img src="http://maps.googleapis.com/maps/api/
staticmap?center='+position.coords.latitude+','+position.coords.longit
ude+'&zoom=12&size=300x300&scale=2&sensor=true">';
  var element = document.getElementById('map_holder');
  element.innerHTML = mapOutput;
}
```

Here you can see that instead of creating the coordinates, the map and the markers as in the earlier code listing, we simply request an image source using the Static Maps API, and send in the coordinates, image size, and other data as parameters.

By using the Static Map API, you lose the interactivity offered through the dynamic map, but you gain an incredibly simple, easy-to-use service that requires very little code to achieve results.

You can find out more about the Google Static Map API on the official documentation, available here: `https://developers.google.com/maps/documentation/staticmaps/`.

Creating a visual compass to show the devices direction

The PhoneGap API provides developers with the ability to receive coordinate and heading information from the device. We can use this information to build a custom compass tool that responds to the device movement.

How to do it...

1. Create the HTML layout for our page, including the `cordova-2.0.0.js` reference.

```
<!DOCTYPE html>
<html>
  <head>
     <meta name="viewport" content="user-scalable=no,
            initial-scale=1, maximum-scale=1,
            minimum-scale=1, width=device-width;" />

    <title>Compass</title>
    <script type="text/javascript"
        src="cordova-2.0.0.js"></script>
  </head>
  <body>

  </body>
</html>
```

2. In this example, we will be referencing certain elements within the DOM by class name. For this we will use the **XUI** JavaScript library (`http://xuijs.com/`). Add the `script` reference within the `head` tag of the document to include this library.
3. Let's also create the `script` tag block that will hold our custom JavaScript for interaction with the PhoneGap API, as shown in the following code:

```
<head>
   <meta name="viewport" content="user-scalable=no,
           initial-scale=1, maximum-scale=1,
           minimum-scale=1, width=device-width;" />

  <title>Compass</title>
  <script type="text/javascript"
      src="cordova-2.0.0.js"></script>

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


  <script type="text/javascript">
  // PhoneGap code will go here

  </script>

</head>
```

4. Add a new `div` element within the `body` tag and give this the `class` attribute of `container`. This will hold our compass elements for display.
5. The compass itself will be made up of two images. Both images will have an individual `class` name assigned to them, which will allow us to easily reference each of them within the JavaScript. Add these two within the `container` element.
6. Next, write a new `div` element below the images with the `id` attribute set to `heading`. This will hold the text output from the compass response:

```
<body>

  <div class="container">

      <img src="images/rose.png"
        class="rose" width="120" height="121"
        alt="rose" />

      <img src="images/compass.png"
        class="compass" width="200" height="200"
        alt="compass" />

    <div id="heading"></div>

  </div>

</body>
```

7. With the initial layout complete, let's start writing our custom JavaScript code. First, let's define the `deviceready` event listener. As we are using `XUI`, this differs a little from other recipes within this chapter:

```
Var headingDiv;

x$(document).on("deviceready", function () {

});
```

8. When we have a result to output to the user of the application, we want the data to be inserted into the `div` tag with the `heading id` attribute. `XUI` makes this a simple task, and so we'll update the `headingDiv` global variable to store this reference:

```
x$(document).on("deviceready", function () {
  headingDiv = x$("#heading");

});
```

9. Let's now include the requests to the PhoneGap compass methods. We'll actually call two requests within the same function. First we'll obtain the current heading of the device for instant data, then we'll make a request to watch the device heading, making the request every tenth of a second thanks to the use of the `frequency` parameter. This will provide use with continual updates to ensure the compass is correct:

```
navigator.compass.getCurrentHeading(onSuccess, onError);
navigator.compass.watchHeading(
              onSuccess, onError, {frequency: 100});
```

10. Both of these requests use the same `onSuccess` and `onError` method to handle output and data management. The `onSuccess` method will provide us with the returned data in the form of a `heading` object.
11. We can use this returned data to set the HTML content of the heading element with the generated message string, using the `headingDiv` variable we defined earlier.
12. Our visual compass also needs to respond to the heading information. Using the `CSS` method from `XUI`, we can alter the `transform` properties of the rose image to rotate using the returned `magneticHeading` property. Here we reference the image by calling its individual class name, `.rose`:

```
// Run after successful transaction
// Let's display the compass data
function onSuccess(heading) {
  headingDiv.html(
    'Heading: ' + heading.magneticHeading + '&#xb0; ' +
    convertToText(heading.magneticHeading) + '<br />' +
    'True Heading: ' + heading.trueHeading + '<br />' +
    'Accuracy: ' + heading.headingAccuracy
      );

  // Alter the CSS properties to rotate the rose image
  x$(".rose").css({
    "-webkit-transform":
    "rotate(-" + heading.magneticHeading + "deg)",
     "transform":
    "rotate(-" + heading.magneticHeading + "deg)"
  });

}
```

13. With the `onSuccess` handler in place, we now need to add our `onError` method to output a user-friendly message, should we encounter any problems obtaining the information, as shown in the following code:

```
// Run if we face an error
// obtaining the compass data
function onError() {
  headingDiv.html(
    'There was an error trying to ' +
    'locate your current bearing.'
  );
}
```

14. When creating our message string in the `onSuccess` function we made a call to a new function called `convertToText`. This accepts the `magneticHeading` value from the `heading` object and converts it into a text representation of the direction for display. Let's include this function outside of the `XUI deviceready` block:

```
// Accept the magneticHeading value
// and convert into a text representation
function convertToText(mh) {
  var textDirection;
  if (typeof mh !== "number") {
    textDirection = '';
  } else if (mh >= 337.5 || (mh >= 0 &&  mh <= 22.5)) {
    textDirection =  'N';
  } else if (mh >= 22.5 && mh <= 67.5) {
     textDirection =  'NE';
  } else if (mh >= 67.5 && mh <= 112.5) {
     textDirection =  'E';
  } else if (mh >= 112.5 && mh <= 157.5) {
     textDirection =  'SE';
  } else if (mh >= 157.5 && mh <= 202.5) {
     textDirection =  'S';
  } else if (mh >= 202.5 && mh <= 247.5) {
     textDirection =  'SW';
  } else if (mh >= 247.5 && mh <= 292.5) {
     textDirection =  'W';
  } else if (mh >= 292.5 && mh <= 337.5) {
     textDirection =  'NW';
  } else {
    textDirection =  textDirection;
  }
  return textDirection;
}
```

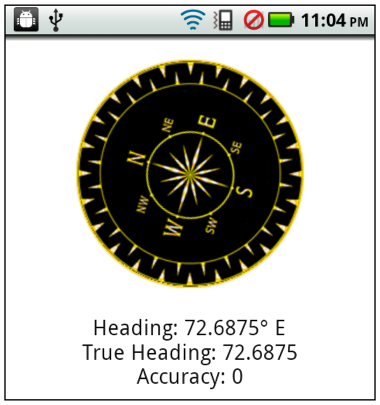
15. Let's provide some CSS to position our two images on the screen and ensure the rose image is overlaying the compass image. Create a new file called `compass_style.css` and insert the following styles into it:

```
.container {
  position: relative;
  margin: 0 auto;
  width: 200px;
  overflow: hidden;
}

#heading {
  position: relative;
  font-size: 24px;
  font-weight: 200;
  text-shadow: 0 -1px 0 #eee;
  margin: 20px auto 20px auto;
  color: #111;
  text-align: center;
}
.compass {
  padding-top: 12px;
}
.rose {
  position: absolute;
  top: 53px;
  left: 40px;
  width: 120px;
  height: 121px;
}
```

16. Finally, include the reference to the `compass_style.css` file in the `head` tag of the HTML document:

```
<title>Compass</title>
<link href="compass_style.css"
    rel="stylesheet" />
<script type="text/javascript"
    src="cordova-2.0.0.js"></script>
<script type="text/javascript"
    src="xui.js"></script>
```

17. Running the application on the device, the output will look something like the following screenshot:



Heading: 72.6875° E
True Heading: 72.6875
Accuracy: 0

How it works...

The `watchHeading` method from the PhoneGap API compass functionality retrieves periodic updates containing the current heading of the device at the interval specified as the value of the `frequency` variable passed through. If no interval is declared, a default value of 100 milliseconds (one-tenth of a second) is used.

With every successful request made on the continuous cycle, the `onSuccess` method is executed and formats the data for output onto the screen, as well as making a change to the `transform` property of the graphical element to rotate in accordance with the heading.

The `onSuccess` method returns the obtained heading information in the form of the `compassHeading` object, which contains the following properties:

- `magneticHeading`: A `Number` value ranging from 0 to 359.99 that specifies a heading in degrees at a single moment in time.
- `trueHeading`: A `Number` value ranging from 0 to 359.99 that specifies the heading relative to the geographic North Pole in degrees.
- `headingAccuracy`: A `Number` value that indicates any deviation in degrees between the reported heading and the true heading values.
- `timestamp`: The time in milliseconds at which the heading was determined.

See also

- *Chapter 6, Working with XUI*