Chapter 8

Using PhoneGap API

In this chapter, we cover:

- Using Accelerometer API
- Using Camera API
- Using Capture API—Capturing audio and video
- Using Geolocation API—Getting a device's current location and watching a device's location
- Using Google Maps API—Displaying Google Maps
- Combining the Geolocation and Google Maps APIs—Displaying Google Maps centered at the device location, displaying markers on the map, drawing lines on the map, finding the distance between two positions, dragging markers, and computing distances
- Using the Geocoder class
- Using Compass API

Using Accelerometer API

An accelerometer is a sensor that provides information about the current position of the device, detecting its motion, tilt, and acceleration. This sensor is available on most modern smartphones. Using the accelerometer, we can develop applications that respond to the position or motion of the device. PhoneGap provides the Accelerometer API that enables us to access the accelerometer sensor of the device and use its information in Web applications.

To access the accelerometer sensor in Web applications, we need to first create an `Acceleration` object. The `Acceleration` object is a read-only object that contains accelerometer data, that is, device motion in the X, Y, and Z directions at a specific time. Below are the `Acceleration` object's properties:

- `x`—Represents the amount of acceleration along the X axis.
- `y`—Represents the amount of acceleration along the Y axis.
- `z`—Represents the amount of acceleration along the Z axis.
- `timestamp`—Timestamp for creation, is expressed in milliseconds.

Using Accelerometer Methods

Let us learn about the methods that are required in the creation of the `Acceleration` object. Below is the first one.

`getCurrentAcceleration()` Method

An `Acceleration` object can be created by calling `getCurrentAcceleration()`, as shown in the example below:

```
navigator.accelerometer.getCurrentAcceleration(onSuccess, onError);
```

The method above, if executed successfully, will create an `Acceleration` object that can be accessed from the `onSuccess` callback method, as shown below:

```
function onSuccess(acceleration) {
   alert('X Coordinate: ' + acceleration.x + '\n' +
   'Y Coordinate: ' + acceleration.y + '\n' +
   'Acceleration Z: ' + acceleration.z + '\n' +
   'Timestamp: ' + acceleration.timestamp + '\n');
};
```

We can see that the `Acceleration` object's `x`, `y`, `z`, and `timestamp` properties are used for displaying acceleration of the device along the three axes, along with the timestamp. In case some error occurs while creating an `Acceleration` object, the `onError` callback method will be called to display the error message, as shown below:

```
function onError() {
   alert('Error occurred while using Accelerometer');
};
```

The `getCurrentAcceleration()` method is called every time there is a change in the acceleration of the device; hence, it results in high consumption of CPU and battery power. To avoid consumption of CPU, the `watchAcceleration()` method is preferred. Let us now learn more.

`watchAcceleration()` Method

The `watchAcceleration()` method watches or gets the acceleration data of the device in the specified time interval. For example, the following statements will watch the device acceleration after every second:

```
var options = {frequency: 1000};
var watchID = navigator.accelerometer.watchAcceleration(onSuccess,
onError, options);
```

We can see that in addition to the two callback methods, `onSuccess` and `onError`, the `watchAcceleration()` method uses an `options` object with a `frequency` parameter. The `frequency` parameter specifies the time interval between watching and measuring of acceleration. The time interval is specified in milliseconds.

clearWatch() Method

To stop watching or getting the acceleration data through the `Acceleration` object created earlier, the `clearWatch()` method is used. The reference or ID of the `Acceleration` object is passed to this method as shown below:

```
navigator.accelerometer.clearWatch(watchID);
```

The statement above stops the measuring of acceleration data through the `Acceleration` object of the ID `watchID`.

Let us apply the knowledge gained so far in measuring acceleration of an Android device. So, create an Android project called `PGAccelerometerApp` and configure it for using PhoneGap. In the `assets/www` folder, open the `index.html` file and write the code as shown in Listing 8.1.

Listing 8.1 Code Written in the `index.html` File

```
<!DOCTYPE HTML>
<html>
   <head>
   <title>PhoneGap Application</title>
   <script type = "text/javascript" charset = "utf-8" src = "cordova-2.3.0.js">
   </script>
   <script type = "text/javascript">
   function onBodyLoad() {
      document.addEventListener("deviceready", PhonegapLoaded, false);
   }
   function PhonegapLoaded(){
      var options = {frequency: 500};
      navigator.accelerometer.watchAcceleration(onSuccess, onError, options);
   }
   function onSuccess(acceleration) {
      Xaxis = document.getElementById("Xaxis");
      Yaxis = document.getElementById("Yaxis");
      Zaxis = document.getElementById("Zaxis");
      Xaxis.innerHTML = "X Coordinate: " + acceleration.x;
      Yaxis.innerHTML = "Y Coordinate: " + acceleration.y;
      Zaxis.innerHTML = "Z Coordinate: " + acceleration.z;
   }
   function onError() {
      alert("Error occurred while using Accelerometer");
   }
   </script>
   </head>
   <body onload = "onBodyLoad()">
      <h3>Accelerometer</h3>
      <div id = "Xaxis"></div>
      <div id = "Yaxis"></div>
      <div id = "Zaxis"></div>
      </form>
   </body>
</html>
```

In the code above, we can see that three `<div>` elements of IDs `Xaxis`, `Yaxis`, and `Zaxis` are defined that will be used to display the acceleration data of the device along the three axes,

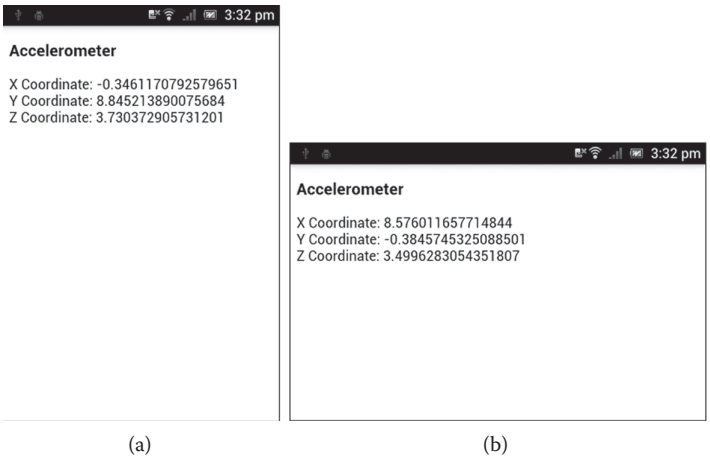


Figure 8.1 (a) The X, Y, and Z coordinate values when the device is in portrait mode. (b) The coordinates along the three axes when the device switches to landscape mode.

respectively. After PhoneGap is loaded, the `options` object of the `watchAcceleration()` method is defined to measure the acceleration data of the device every half second, that is, every 500 milliseconds. The `onSuccess` callback method will be called if no error occurs in the execution of the `watchAcceleration()` method. In the `onSuccess` callback method, the three `<div>` elements with IDs `Xaxis`, `Yaxis`, and `Zaxis` are accessed and the acceleration data along the three axes are displayed through them. If anything goes wrong while accessing the acceleration data, the `onError` callback method is called to display the error message through the alert dialog.

Upon running the application when the device is in portrait mode, the acceleration data along the X, Y, and Z axes will be displayed as shown in Figure 8.1a. Upon switching the device to landscape mode, the acceleration data along the three axes will appear as shown in Figure 8.1b.

Using Camera API

Every smartphone comes with a camera, and it is quite obvious that developers need to access and use it in applications. The PhoneGap Camera API enables Web applications to access the device camera and the images in the local photo storage of the device.

To obtain or access a picture using the device camera, the `getPicture()` method is called as shown below:

```
navigator.camera.getPicture(onCameraSuccess, onCameraError,
cameraOptions);
```

The `onCameraSuccess` callback method is called when the image is obtained successfully. The `getPicture()` method returns either the Uniform Resource Identifier (URI) pointing to the image file on the device's file system or the base64-encoded string representing the content of the image. The `onCameraError` callback method is called when either the task of obtaining an image is canceled or some error occurs while obtaining the image. The `cameraOptions` object

is used to define parameters that configure the obtained image, its format, and so on. The `cameraOptions` object supports the properties as shown in Table 8.1.

The following shows a sample code to define `cameraOptions` and the calling `getPicture()` method:

Table 8.1 Properties Supported by the `CameraOptions` Object

Property	*Description*
`Quality`	Represents a value in percentage that determines the quality of the image. 100% quality means the retrieved image will be of the highest quality and is not reduced or compressed. Because the file size of the highest-quality image becomes quite large, it is recommended to set the quality percentage at around 50%.
`destinationType`	Determines how the image information is returned. Following are the two possible values for this parameter: • `Camera.DestinationType.FILE_URI`—Provides the file URI or path to the image file in the device's local file system. • `Camera.DestinationType.DATA_URL`—Provides the binary data of the image in base64-encoded string format. It is very easy to work with file URIs compared with the raw binary image data.
`sourceType`	Defines the source of the picture to capture or retrieve, that is, whether to retrieve the image through a device camera or from a saved photo album. The possible values of this parameter are `Camera.SourceType.CAMERA`, `Camera.SourceType.PHOTOLIBRARY`, and `Camera.SourceType.SAVEDPHOTOALBUM`. In most of the mobile platforms, both `PHOTOLIBRARY` and `SAVEDPHOTOALBUM` refer to the photo album.
`allowEdit`	Boolean value that determines if editing can be applied to the retrieved image before sending it to the application. For example, if this parameter is set to `true`, an editing screen will be launched that can be used to edit the image. This parameter works only on the iPhone.
`encodingType`	Determines the kind of picture to return, i.e., whether we want the JPEG or PNG format of the retrieved image. Below are the possible values for this parameter: • `Camera.EncodingType.JPEG`—Used to return an image in JPEG format. It is the most commonly used encoding type. • `Camera.EncodingType.PNG`—Used to return an image in PNG format. This type is not supported on all platforms.
`targetWidth` and `targetHeight`	Determines the height and width of the retrieved image in pixels. The image will be scaled to the specified width and height maintaining the aspect ratio.

(Continued)

Table 8.1 Properties Supported by the `CameraOptions` Object (Continued)

Property	*Description*
`mediaType`	Determines the media type. Below are the possible options: • `DEFAULT`—Returns the image using the format specified in the `destinationType` parameter. • `ALLMEDIA`—Enable selection from all media types. • `PICTURE`—Returns the photographs only. • `VIDEO`—Returns the video files only. When the `VIDEO` option is chosen, only a file URI is returned to the calling application.

```
var cameraOptions = {
   quality : 50,
   sourceType : Camera.PictureSourceType.CAMERA,
   destinationType : Camera.DestinationType.FILE_URI,
   allowEdit : true,
   encodingType: Camera.EncodingType.JPEG,
   targetWidth: 100,
   targetHeight: 200};
   navigator.camera.getPicture(onSuccess, onFail, cameraOptions);
   function onSuccess(data) {
   capturedImage = document.getElementById("capturedImage");
   capturedImage.src = data;
}
function onFail(err) {
   alert('Command Cancelled or Error occurred while retrieving the image:
   ' + err);
}
```

Let us create an application to see how PhoneGap's Camera API is used for invoking the device's default camera application to access the image(s). So, create an Android project called `PGCameraApp` and configure it for using PhoneGap. In the `assets/www` folder, open the `index.html` file and write the code as shown in Listing 8.2.

Listing 8.2 Code Written in the `index.html` File

```
<!DOCTYPE HTML>
<html>
   <head>
   <title>PhoneGap Application</title>
   <script type = "text/javascript" charset = "utf-8" src = "cordova-2.3.0.js">
   </script>
   <script type = "text/javascript">
   function onBodyLoad() {
      document.addEventListener("deviceready", PhonegapLoaded, false);
   }
   function PhonegapLoaded(){
      document.getElementById("captureImage").addEventListener("click",
      captureImage);
   }
   function captureImage(){
```

```
      var cameraOptions = {
         quality : 50,
         sourceType : Camera.PictureSourceType.CAMERA,
         destinationType : Camera.DestinationType.FILE_URI,
         encodingType: Camera.EncodingType.JPEG,
         targetWidth: 100,
         targetHeight: 200};
         navigator.camera.getPicture(displayImage, onError, cameraOptions);
   }
   function displayImage(data){
      capturedImage = document.getElementById("capturedImage");
      capturedImage.style.display = 'block';
      capturedImage.src = data;
   }
   function onError(err) {
      alert('Some error occurred while capturing image: ' + err);
   }
   </script>
   </head>
   <body onload = "onBodyLoad()">
      <button id = "captureImage"onclick = "captureImage();">Capture Image</
      button><br/>
      <img id = "capturedImage" style = "display:none;width:400px;height:300px;"
></img>
   </body>
</html>
```

In the code above, we can see a button with the caption `Capture Image` is defined along with an `<img>` element. The `Capture Image` button, when clicked, will call the JavaScript function `captureImage()` to initiate the task of obtaining an image using the device's camera application. The `<img>` element is assigned the ID `capturedImage` and will be used to display the obtained image.

In the JavaScript function `captureImage()`, the `cameraOptions` object is defined. The properties in `cameraOptions` recommend the quality of the obtained image to 50%, return the file URI of the obtained image, return the image in JPEG format, and scale the obtained image to 100 pixels wide by 200 pixels high, maintaining the aspect ratio. After setting the `cameraOptions` object, the `getPicture()` method is called. The `displayImage` callback method will be called if the image is obtained successfully. In the `displayImage` callback method, the `<img>` element with the ID `capturedImage` is accessed and the URI of the obtained image is assigned to it for display.

The `onError` callback will be called to display an error message if any error occurs while obtaining the image.

Next, we need to write permissions in the `AndroidManifest.xml` file to access the device camera, use it, and store the obtained image in the device storage. Listing 8.3 shows the code in the `AndroidManifest.xml` file. Only the code in bold is new; the rest is the default code.

Listing 8.3 Code in the `AndroidManifest.xml` File

```
<?xml version = "1.0" encoding = "utf-8"?>
   <manifest xmlns:android = "http://schemas.android.com/apk/res/android"
   package = "com.phonegap.pgcameraapp"
   android:versionCode = "1"
   android:versionName = "1.0" >
   <uses-sdk
```

```
      android:minSdkVersion = "11"
      android:targetSdkVersion = "17"/>
   <uses-permission
      android:name = "android.permission.ACCESS_NETWORK_STATE"/>
   <uses-feature android:name = "android.hardware.camera"/>
   <uses-permission android:name = "android.permission.CAMERA"/>
   <uses-permission
      android:name = "android.permission.WRITE_EXTERNAL_STORAGE"/>
   <application
      android:allowBackup = "true"
      android:icon = "@drawable/ic_launcher"
      android:label = "@string/app_name"
      android:theme = "@style/AppTheme" >
      <activity android:name = "com.phonegap.pgcameraapp.PGCameraAppActivity"
         android:label = "@string/app_name" >
         <intent-filter>
            <action android:name = "android.intent.action.MAIN"/>
            <category android:name = "android.intent.category.LAUNCHER"/>
         </intent-filter>
      </activity>
   </application>
</manifest>
```

Our application is ready to run. Upon running the application, a button, `Capture Image`, will be displayed as shown in Figure 8.2a. Upon clicking the `Capture Image` button, the standard camera application will open, enabling us to take a picture (see Figure 8.2b). The clicked image will be displayed in the image element of the Hypertext Markup Language (HTML) page as shown in Figure 8.2c.

Using the Capture API

The PhoneGap Capture API allows an application to capture audio, video, and images using the built-in application on a mobile device. The device's default camera application is used to capture pictures and videos, while the device's default voice recorder application is used for capturing audio clips. The Capture API interacts with the device's default capture application and enables multiple captures with a single API call.

Note: The Camera API can capture only images, but supports alternate sources for the image files. The Camera API is supported by PhoneGap for backward compatibility.

The methods to capture audio, image, and video are provided below:

- To capture one or more audio files, the `captureAudio()` method is used as shown below:

```
navigator.device.capture.captureAudio(CaptureSuccess, CaptureError,
captureOptions);
```

- To capture one or more image files, the `captureImage()` method is used as shown below:

```
navigator.device.capture.captureImage(onCaptureSuccess, onCaptureEr-
ror, captureOptions);
```

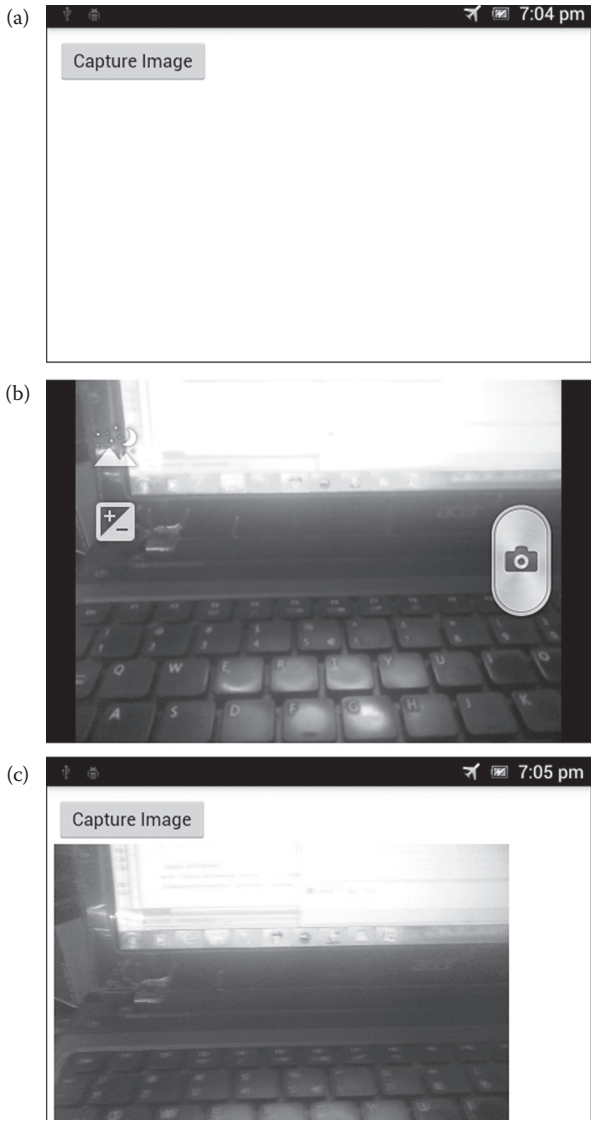


Figure 8.2 (a) The `Capture Image` button appears upon application start-up. (b) The standard camera application is invoked to click images. (c) The clicked image appears in the `<img>` element of the HTML page.

- To capture one or more video files, the `captureVideo()` method is used as shown below:

```
navigator.device.capture.captureVideo(onCaptureSuccess, onCaptureEr-
ror, captureOptions);
```

Let us understand the concept of capturing audio, video, and image one by one. We will begin with capturing audio.

Capturing Audio

To capture one or more audio files, the `captureAudio()` method is called as shown below:

```
navigator.device.capture.captureAudio(captureAudioSuccess, captureAudio-
Error, captureOptions);
```

The `captureAudioSuccess` callback method will be called after the audio is captured by the capture application on the device. The `captureAudioError` callback method is called when either the capturing task is canceled or some error occurs while capturing. The `capture-Options` object supports the parameters that can be used to configure the capturing task. The list of parameters supported by the `captureOptions` object is given below:

- **Limit**—Determines the number of audio clips to capture. This value must be greater than or equal to 1. The default value is 1.
- **Duration**—Determines the maximum duration of audio clips in seconds.
- **Mode**—Determines the mode of capturing audio, such as audio/wav or audio/amr. Wav and amr are the audio formats.

For example, the statements given below capture two audio clips of 10 seconds duration:

```
var captureOptions = {limit:2, duration:10};
navigator.device.capture.captureAudio(captureAudioSuccess, captureAudio-
Error, captureOptions);
```

If audio is captured successfully, the `captureAudioSuccess` callback will be called; otherwise, the `captureAudioError` callback will be called. An array of media files containing information about the captured audio will be passed to the `captureAudioSuccess` callback method. The media file array passed to the function supports the following properties:

- `name`—Represents the name of the file along with the extension.
- `fullPath`—Represents the full path of the media file.
- `type`—Represents the file's Multipurpose Internet Mail Extensions (MIME) type.
- `lastModifiedDate`—Represents the date and time the media file was last modified.
- `size`—Represents the file's size in bytes.

The `captureAudioSuccess` callback method loops through the media file array and processes each of the media files generated during the capture as shown below:

```
function captureAudioSuccess(mediaFiles) {
   var len, i;
   len = mediaFiles.length;
   if(len > 0) {
      for(i = 0; i < len; i + = 1) {
         alert("File: "+mediaFiles[i].name+ "\n"+
         "is stored at: "+mediaFiles[i].fullPath+"\n"+
         "Size of the file is: "+mediaFiles[i].size);
      }
   } else {
      alert("Error occurred while capturing audio");
   }
}
```

We see that the `captureAudioSuccess` callback loops through the media file array, isolates each media file, and displays its name, path, and size. The `captureAudioError` callback method is called to display an error message when either the capturing task is canceled or some error occurs, as shown below:

```
function captureAudioError(err) {
   alert("Error occurred while recording audio:" +err);
}
```

Let us apply the knowledge gained so far in creating an application that uses the device's default voice recorder to capture audio(s). Create an Android project called `PGMediaApp` and configure it for using PhoneGap. In the `assets/www` folder, open the `index.html` file and write the code as shown in Listing 8.4.

Listing 8.4 Code Written in the `index.html` File

```
<!DOCTYPE HTML>
<html>
   <head>
   <title>PhoneGap Application</title>
   <script type = "text/javascript" charset = "utf-8" src = "cordova-2.3.0.js">
   </script>
   <script type = "text/javascript">
   function onBodyLoad() {
      document.addEventListener("deviceready", PhonegapLoaded, false);
   }
   function PhonegapLoaded(){ document.getElementById("recordAudio").
   addEventListener("click", captureAudio);
   }
   function captureAudio() { navigator.device.capture.captureAudio
   (captureAudioSuccess, captureAudioError, {limit: 1});
   }
   function captureAudioSuccess(mediaFiles) {
      var len, i;
      len = mediaFiles.length;
      if(len > 0) {
         for(i = 0; i < len; i + = 1) {
            alert("Audio recorded successfully"+"\n"+
            "File: "+mediaFiles[i].name+ "\n"+
            "is stored at: "+mediaFiles[i].fullPath+"\n"+
            "Size of the file is: "+mediaFiles[i].size);
         }
      }
   }
   function captureAudioError(err) {
      alert("Error occurred while recording audio:" +err);
   }
   </script>
   </head>
   <body onload = "onBodyLoad()">
      <button id = "recordAudio">Capture Audio</button>
   </body>
</html>
```

We can see that a button with the ID `recordAudio` and caption `Capture Audio` is defined in the `<body>` of the HTML file. The `Capture Audio` button, when clicked, invokes

the captureAudio() method, which in turn invokes the device's default audio recorder. If the audio is recorded successfully, the captureAudioSuccess callback method is called to display the filename of the recorded audio path where it is stored on the device and the file size. If some error occurs while capturing the audio, the captureAudioError callback is called to display the reason for the error on the screen.

To access the device's default voice recorder, record audio, and save the recorded audio in the device storage, add the following permission statements to the <manifest> element in the AndroidManifest.xml file:

```
<uses-permission android:name = "android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name = "android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name = "android.permission.RECORD_AUDIO"/>
<uses-permission android:name = "android.permission.MODIFY_AUDIO_SETTINGS"/>
```

Upon running the application, a button, Capture Audio, will appear as shown in Figure 8.3a. Upon clicking the Capture Audio button, the standard audio application in the device will open, enabling us to record and play audio (see Figure 8.3b). Upon clicking the record button, the application will begin recording the user's voice as shown in Figure 8.3c. Upon clicking the Stop button, the audio recording will stop. Two options will appear, Discard and Done (see Figure 8.3d). The Discard button will cancel the recorded audio. The Done button, if clicked, will save the recorded audio in the device storage and will appear in the My recordings list of the device. Upon clicking the Done button, an alert dialog will appear, informing us that the audio recorded successfully along with the audio filename, its path, and file size, as shown in Figure 8.3e. The recorded audio will appear in the My recording list of the device (see Figure 8.3f).

Capturing Video

To capture video on a device, the captureVideo() method is used as shown below:

```
navigator.device.capture.captureVideo(captureVideoSuccess,
captureVideoError, captureOptions);
```

The captureVideoSuccess callback is called if the video is captured successfully. An array of media file objects, each describing the captured video clip file, is passed to the callback. The media file array supports the same properties, name, fullPath, type, lastModificationDate, and size that we saw in the captureAudio() method.

The captureVideoError callback is called if the capture process is canceled or some error occurs while capturing video. The captureOptions object defines the parameters to configure video capturing. The following are the parameters supported by the captureOptions object:

- **Limit**—Represents the maximum number of video clips that can be captured at a time. The value of this parameter must be greater than or equal to 1. Its default value is 1.
- **Duration**—Represents the maximum duration of the video clip in seconds.
- **Mode**—Represents the mode of video capture, like video/quicktime, video/3gpp, and so on.

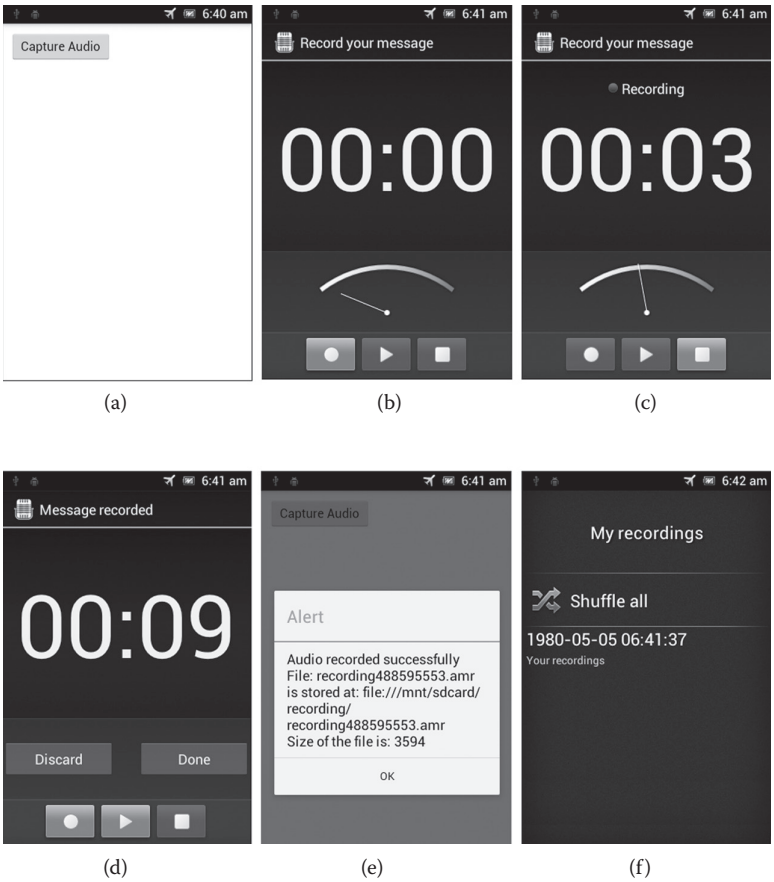


Figure 8.3 (a) `Capture Audio` appears on application start-up. (b) The standard audio application in the device is invoked. (c) Recording audio. (d) Audio recording stopped. (e) Alert dialog informing us that `audio recorded successfully`. (f) The recorded audio appears in the `My recordings` list of the device.

Below are example captures of a video clip that is 30 seconds in duration:

```
var captureOptions = {limit: 1, duration: 30};
navigator.device.capture.captureVideo(captureVideoSuccess,
captureVideoError, captureOptions);
function captureVideoSuccess(mediaFiles) {
   var len, i;
   len = mediaFiles.length;
   if(len > 0) {
      for(i = 0; i < len; i + = 1) {
         alert("Video Clip: "+mediaFiles[i].name+ "\n"+
         "is stored at: "+mediaFiles[i].fullPath+"\n"+
         "Size of the video clip is: "+mediaFiles[i].size);
      }
   }
}
```

```
function captureVideoError(err) {
   alert("Error occurred while recording video:" +err);
}
```

The captureVideoSuccess callback will be called if the captureVideo() method executes successfully; that is, the device's default application successfully records the video(s). The media file array of the recorded video clips is passed to the captureVideoSuccess callback where the filename, path, and size of each video recorded clip are displayed.

The PGMediaApp that we just created earlier captures audio. Let us modify the same application to record video too. That is, in addition to the Capture Audio button, we will add one more button, Capture Video, to the application. When the user clicks the Capture Video button, the device's video recorder application will be invoked to record the video clip(s). So, open the Android project, PGMediaApp. Open the index.html file in the assets/www folder and modify its code to appear as shown in Listing 8.5. Only the code in bold is modified; the rest is the same as we saw in Listing 8.4.

Listing 8.5 Code Written in the index.html File

```
<!DOCTYPE HTML>
<html>
   <head>
   <title>PhoneGap Application</title>
   <script type = "text/javascript" charset = "utf-8" src = "cordova-2.3.0.js">
   </script>
   <script type = "text/javascript">
   function onBodyLoad() {
      document.addEventListener("deviceready", PhonegapLoaded, false);
   }
   function PhonegapLoaded(){ document.getElementById("recordAudio").
   addEventListener("click", captureAudio); document.
   getElementById("recordVideo").addEventListener("click", captureVideo);
   }
   function captureAudio() {
      navigator.device.capture.captureAudio(captureAudioSuccess,
      captureAudioError, {limit: 1});
   }
   function captureAudioSuccess(mediaFiles) {
      var len, i;
      len = mediaFiles.length;
      if(len > 0) {
         for(i = 0; i < len; i + = 1) {
            alert("Audio recorded successfully"+"\n"+
            "File: "+mediaFiles[i].name+ "\n"+
            "is stored at: "+mediaFiles[i].fullPath+"\n"+
            "Size of the file is: "+mediaFiles[i].size);
         }
      }
   }
   function captureAudioError(err) {
      alert("Error occurred while recording audio:" +err);
   }
   function captureVideo() { navigator.device.capture.captureVideo
   (captureVideoSuccess, captureVideoError, {limit: 1});
   }
   function captureVideoSuccess(mediaFiles) {
      var len, i;
```

```
        len = mediaFiles.length;
        if(len > 0) {
           for(i = 0; i < len; i + = 1) {
              alert("Video recorded successfully" + "\n" +
              "Video Clip filename: " + mediaFiles[i].name + "\n"+
              "is stored at: " + mediaFiles[i].fullPath + "\n" +
              "Size of the video clip is: " + mediaFiles[i].size);
           }
        }
        else
           alert("Video is not recorded");
     }
     function captureVideoError(err) {
        alert("Error occurred while recording video:" +err);
     }
     </script>
     </head>
     <body onload = "onBodyLoad()">
        <button id = "recordAudio">Capture Audio</button>   
        <button id = "recordVideo">Capture Video</button>
     </body>
</html>
```

We can see that a button with the ID `recordVideo` and caption `Capture Video` is added in the `<body>` of the HTML file. The `Capture Video` button, when clicked, invokes the `captureVideo()` method, which in turn invokes the device's camera application to record video. If the video is recorded successfully, the `captureVideoSuccess` callback method is called to display detailed information of the recorded video, for example, the filename of the video, path where it is stored on the device, and file size. If some error occurs while capturing the video, the `captureVideoError` callback is called to display the error that occurred while recording the video.

To access the device's camera to record video, we need to add the following permission statement in the `<manifest>` element in the `AndroidManifest.xml` file:

```
<uses-permission android:name = "android.permission.CAMERA"/>
```

Now our application is ready to run. Upon running the application, we find two buttons, `Capture Audio` and `Capture Video`, as shown in Figure 8.4a. Upon clicking the `Capture Video` button, the standard camera application in the device will open, enabling us to record video (see Figure 8.4b). Upon clicking the record button, video will begin recording, as shown in Figure 8.4c. Upon clicking the stop button, video recording will stop. The recorded video clip will be saved in the storage of the device. An alert dialog informing us of the successful recording of the video along with the filename, its path, and size will appear as shown in Figure 8.4d. The recorded video will appear in the storage of the device (see Figure 8.4e).

Capturing Images

Earlier, we saw how the Camera API can be used for retrieving images. Let us now look at how the images are captured using the Capture API. To capture an image on a device, the `captureImage()` method is used as shown below:

```
navigator.device.capture.captureImage(captureImageSuccess,
captureImageError, captureOptions);
```

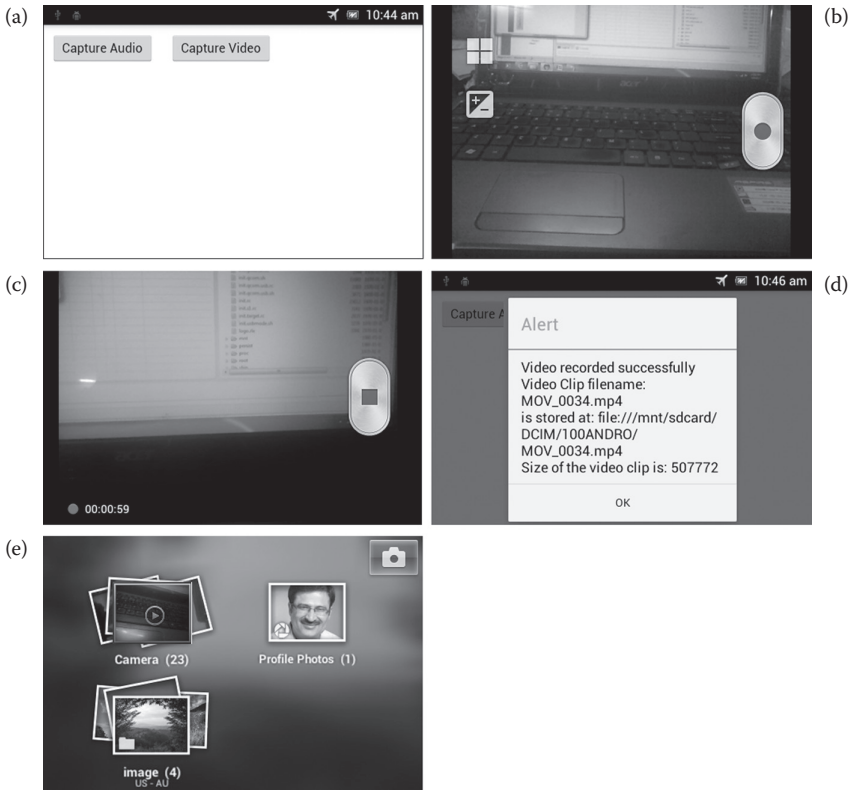


Figure 8.4 (a) `Capture Audio` **and** `Capture Video` **buttons appear on application start-up. (b) The standard camera application in the device is invoked. (c) Recording video. (d) Alert dialog informing us that** `Video recorded successfully along with other information`**. (e) Recorded video appears in the Pictures Library.**

The `captureImageSuccess` callback is called if the image is captured successfully. An array of media file objects, each describing the captured image file(s), is passed to the callback. The media file array supports the same properties, `name`, `fullPath`, `type`, `lastModificationDate`, and `size` that we saw in the `captureAudio()` method.

The `captureImageError` callback is called if the capture process is canceled or some error occurs while capturing the image. The `captureOptions` object defines the parameters to configure image capturing—the same as we saw in the `captureVideo()` method.

The example given below captures an image using the device's default camera application:

```
<!DOCTYPE HTML>
<html>
   <head>
   <title>PhoneGap Application</title>
   <script type = "text/javascript" charset = "utf-8" src = "cordova-
   2.3.0.js"></script>
   <script type = "text/javascript">
   function onBodyLoad() {
      document.addEventListener("deviceready", PhonegapLoaded, false);
```

```
    }
    function PhonegapLoaded(){
    document.getElementById("captureImage").addEventListener("click",
    captureImage);
    }
    function captureImage() {
    navigator.device.capture.captureImage(captureImageSuccess,
    captureImageError, {limit: 1});
}
function captureImageSuccess(mediaFiles) {
var len, i;
    len = mediaFiles.length;
    if(len > 0) {
       for(i = 0; i < len; i + = 1) {
          alert("Image captured successfully"+"\n"+
          "Image File: "+mediaFiles[i].name+ "\n"+
          "is stored at: "+mediaFiles[i].fullPath+"\n"+
          "Size of the image file is: "+mediaFiles[i].size);
       }
    }
}
function captureImageError(err) {
alert("Error occurred while capturing image:" +err);
}
    </script>
    </head>
    <body onload = "onBodyLoad()">
       <button id = "captureImage">Capture Image</button>
    </body>
</html>
```

The code above is self-explanatory and requires no further explanation.

Using Geolocation API

Geolocation API can be used to fetch the location information of the device in terms of latitude and longitude. It is the global positioning system (GPS) that is usually used to determine a device's location, including signals like the IP address, WiFi, and so on. Let us begin by finding the device's current location.

Getting a Device's Current Location

To determine the location of a smartphone, the `getCurrentPosition()` method is used as shown below:

```
navigator.geolocation.getCurrentPosition(onSuccess, onError, geoOptions);
```

The `onSuccess` callback is executed when the device's location is successfully measured, and the `onError` callback is called if some error occurs while measuring the device's location.

The optional `geoOptions` object can be passed to the `getCurrentPosition()` method to configure it. The `geoOptions` object supports the following properties:

- `enableHighAccuracy`—Boolean value that determines if the device's location has to be measured with a higher degree of accuracy. This option, if set to `true`, will give a more accurate value, but at the same time consumes more processing and battery power.
- `maximumAge`—Defines the maximum age in milliseconds of a cached location value. A smaller value for this property will make the `getCurrentPosition()` method update the device location more frequently.
- `timeout`—Defines the maximum amount of time in milliseconds that can be spent between the execution of the `getCurrentPosition()` method call and the `onSuccess` callback method before declaring a timeout.

The following `geoOptions` object configures the `getCurrentPosition()` method to measure the device location with higher accuracy and declare a timeout if a response is not received within 3000 milliseconds, that is, 3 seconds:

```
var geolocationOptions = {
   timeout : 3000,
   enableHighAccuracy : true
};
```

The `position` object is passed to the `onSuccess` callback method that includes the `Coordinates` object, which in turn contains the properties that can be used to display the device's location. The list of properties supported by the `Coordinates` object is provided below:

- `latitude`—Represents the latitude expressed in decimal degrees.
- `longitude`—Represents the longitude expressed in decimal degrees.
- `altitude`—Represents the height above sea level in meters.
- `accuracy`—Represents the accuracy of the latitude/longitude reading in meters.
- `altitudeAccuracy`—Represents the accuracy of the altitude coordinate in meters.
- `heading`—Represents the direction of travel relative to true north in degrees.
- `speed`—Represents the current ground speed in meters per second.

The following code shows how a position object passed to the `onSuccess` callback can be used to display the location information of the device:

```
function onSuccess(position) {
   var element = document.getElementById('geolocationinfo');
   element.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: ' + new Date(position.timestamp) + '<br/>';
}
```

As stated earlier, when an error occurs while measuring the device's location, the `onError` callback method is executed. An `error` object is passed to the `onError` callback whose `code` and `message` properties can be used to display the reason that caused the error. The error code can be one of the following constants:

- `PositionError.PERMISSION_DENIED`
- `PositionError.POSITION_UNAVAILABLE`
- `PositionError.TIMEOUT`

The `message` property of the `error` object defines the cause of the error in detail. An alert dialog can be used in the `onError` callback to display the error message as shown below:

```
function onError(error) {
   alert('code: ' + error.code + '\n' + 'message: ' + error.message + '\n');
}
```

Watching a Device's Location

To watch a device's location periodically, geolocation watch is used. To create a geolocation watch, the `watchPosition()` method is used as shown below:

```
watchID = navigator.geolocation.watchPosition(onSuccess, onError,
geoOptions);
```

The newly created geolocation watch is assigned to the `watchID` variable. The `onSuccess` and `onError` callback methods are respectively called when the device's location is measured successfully and when some error occurs while measuring the device's location. The `geoOptions` object can be used to configure the watch.

The following `geoOptions` object configures the watch to ignore the cached values that are older than 5000 milliseconds (5 seconds), declares a timeout if a response is not received within 2000 milliseconds (2 seconds), and measures the location of the device with higher accuracy:

```
var geoOptions = {
   maximumAge : 5000,
   timeout : 2000,
   enableHighAccuracy : true
};
```

After defining the `geoOptions` object, the `watchPosition()` method can be called as shown below:

```
watchID = navigator.geolocation.watchPosition(onSuccess, onError, geoOp-
tions);
```

For example, we saw in the `getCurrentPosition()` method, as well as in the `watchPosition()` method, that a `position` object is passed to the `onSuccess` callback that contains the same geolocation properties that we discussed in the `getCurrentPosition()` method. The code below displays the location of the device through the `position` object passed to the `onSuccess` callback method:

```
function onSuccess(position) {
   var element = document.getElementById('geolocationinfo');
   element.innerHTML = 'Latitude: ' + position.coords.latitude + '<br/>' +
   'Longitude: ' + position.coords.longitude + '<br/>' +
   'Timestamp: ' + new Date(position.timestamp) + '<br/>';
}
```

In case of any error that might occur while finding the device's location, the `onError` callback is called that displays the reason that caused the error, as shown below:

```
function onError(error) {
   alert('code: ' + error.code + '\n' + 'message: ' + error.message + '\n');
}
```

Canceling a Watch

To cancel the watch, the `clearWatch()` method is called, passing the `watchID` variable that is created through the `watchPosition()` method:

```
navigator.geolocation.clearWatch(watchID);
```

The resources allocated to `watchID` can be released along with a message informing us of the cancelation of the watch, as shown below:

```
function cancelWatch() {
   navigator.geolocation.clearWatch(watchID);
   watchID = null;
   alert("Location Watch Cancelled");
}
```

Using Google Maps API

The Google Maps API is used to integrate the Google Maps service into our Web applications. We need to subscribe to the API's console Web site to obtain the API key. The API key is used in the applications so that the service can monitor the application's usage. The following are the main classes of the Google Maps API:

- `google.maps.Map` class—Defines a single map on a page. Any number of instances of this class can be created where each instance will define a separate map on the page.
- `google.maps.LatLng` class—Represents geographical coordinates in terms of latitude and longitude.
- `google.maps.Marker` class—Used to place markers on a map.
- `Geocoder` class—Used to convert an address to a `LatLng` object, and vice versa.

Displaying Google Maps

Each instance of `google.maps.Map` class displays a simple Google map on a page. The `Map` class accepts the following two parameters:

- An HTML element that will contain the map. The preferred HTML element for displaying a map on a Web page is a `<div>` element. The code below shows a sample `<div>` element that can be used to display a map:

  ```
  <div id = "map_canvas" style = "width:100%; height:100%"></div>
  ```

  In the example above, we define a `<div>` element named `map-canvas` and set its size to `100%`, which will expand it to fit the size on mobile devices.
- The second parameter used in the `Map` class is a `google.maps.MapOptions` object that is used to initialize the map. The `MapOptions` object defines map initialization variables as shown in Table 8.2.

The following example defines the map of type ROADMAP:

```
mapTypeId: google.maps.MapTypeId.ROADMAP
```

Table 8.2 A Brief Description of the Initialization Variables Used in the `MapOptions` Object

Initialization Variables	*Description*
Latitudes and longitudes	To center the map on a specific point, we create a `LatLng` object and pass the desired location's coordinates to it in the order {latitude, longitude}. In the following example, we will center the map at 37.76944 latitude and –122.43444 longitude: `center: new google.maps.LatLng(37.76944, -122.43444)`
Zoom levels	Determine the resolution to display the map. Value 0 for the zoom level will make the map fully zoomed out, and as we increase the value of this property, the map will zoom in at a higher resolution. In the following example, we will zoom in the map at the resolution value 8: `zoom: 8`
Map types	Determine the map type. The following map types are supported: • `ROADMAP`—Displays the normal, default road map view. • `SATELLITE`—Displays Google Earth satellite images. • `HYBRID`—Displays a mix of satellite views along with a layer for roads, city names, etc. • `TERRAIN`—Displays physical map displaying terrain information like mountains, rivers, etc.

A sample MapOptions object may look as given below:

```
var mapOptions = {
   center: new google.maps.LatLng(37.76944, -122.43444),
   zoom: 8,
   mapTypeId: google.maps.MapTypeId.ROADMAP
};
```

To create a new instance of the Map class, the JavaScript new operator is used as shown below:

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

We can see that a new Map instance called map is created. While creating a new map instance, we specify a HTML element in the page that acts as a container for the map. The reference of the element is obtained in JavaScript via the document.getElementById() method.

The Maps JavaScript API is written into a <script> tag as shown below:

```
<script type = "text/javascript" src = "http://maps.google.com/maps/api/
js?sensor = false"></script>
```

The sensor parameter of the URL must be included in the <script> tag, and it indicates whether the application uses a sensor (such as a GPS locator) to determine the user's location. The statement above indicates that no sensor will be used in the application.

The code given below sums up what we learned above. The code displays a Google map of ROADMAP type centered at 37.76944, –122.43444:

```
<!DOCTYPE HTML>
<html>
   <head>
   <title>PhoneGap Application</title>
   <script type = "text/javascript" charset = "utf-8" src = "cordova-
   2.3.0.js"></script>
   <script type = "text/javascript" src = "http://maps.google.com/maps/
   api/js?sensor = false"></script>
   <link href = "http://code.google.com/apis/maps/documentation/
   javascript/examples/default.css" rel = "stylesheet" type = "text/css"/>
   <script type = "text/javascript">
   function onBodyLoad() {
      document.addEventListener("deviceready", PhonegapLoaded, false);
   }
   function PhonegapLoaded(){
      var mapOptions = {
      center: new google.maps.LatLng(37.76944, -122.43444),
      zoom: 8,
      mapTypeId: google.maps.MapTypeId.ROADMAP
   };
   var map = new google.maps.Map(document.getElementById("map_canvas"),
   mapOptions);
   }
```

```
</script>
</head>
<body onload = "onBodyLoad()">
   <div id = "map_canvas" style = "width:100%; height:100%"></div>
</body>
```

In the code above, we ensure that the Maps API JavaScript code is loaded after PhoneGap is fully loaded. The `onBodyLoad()` function executes and loads PhoneGap. Once PhoneGap is loaded, the Maps API JavaScript code is loaded to perform the task of displaying the map.

Combining the Geolocation and Google Maps API

We have learned about the Geolocation and Google Maps APIs, their classes, and methods that are used in finding the location of the device and displaying Google maps, respectively. How about combining the two APIs?

Let us create an application that displays Google maps as well as measures the location of the device. So, launch Eclipse and create a new Android project called `PGGoogleMapApp` and configure it for using PhoneGap. In the `assets/www` folder, open the `index.html` file and write the code as shown in Listing 8.6.

Listing 8.6 Code Written in the `index.html` File

```
<!DOCTYPE HTML>
<html>
   <head>
   <title>PhoneGap Application</title>
   <script type = "text/javascript" charset = "utf-8" src =
   "cordova-2.3.0.js"></script>
   <script type = "text/javascript" src = "http://maps.google.com/maps/api/
   js?sensor = false"></script>
   <link href = "http://code.google.com/apis/maps/documentation/javascript/
   examples/default.css" rel = "stylesheet" type = "text/css"/>
   <script type = "text/javascript">
   function onBodyLoad() {
      document.addEventListener("deviceready", PhonegapLoaded, false);
   }
   function PhonegapLoaded(){
      var mapOptions = {
      center: new google.maps.LatLng(37.76944, -122.43444),
      zoom: 8,
      mapTypeId: google.maps.MapTypeId.ROADMAP
   };
   var map = new google.maps.Map(document.getElementById("map_canvas"),
   mapOptions);
   navigator.geolocation.watchPosition(onSuccess, onError, {enableHighAccuracy:
   true});
   }
   function onSuccess(position) {
      var element = document.getElementById('geolocationinfo');
      element.innerHTML = 'Latitude: ' + position.coords.latitude + '<br/>' +
      'Longitude: ' + position.coords.longitude + '<br/>' +
      'Timestamp: ' + new Date(position.timestamp) + '<br/>';
   }
```

```
   function onError(error) {
      alert('code: ' + error.code + '\n' + 'message: ' + error.message + '\n');
   }
   </script>
   </head>
   <body onload = "onBodyLoad()">
      <div id = "geolocationinfo">GeoLocation Information</div>
      <div id = "map_canvas" style = "width:100%; height:100%"></div>
   </body>
</html>
```

In the code above, we see that two elements are defined with the IDs geolocationinfo and map _ canvas, respectively. The first element will be used for displaying the geolocation information of the device, and the second element will be used for displaying Google maps. The mapOptions object is defined to make the Google map centered at latitude 37.76944 and longitude -122.43444. Also, the Google map is set to display the default road map view that is zoomed in at level 8. An instance of the Map class is created to display the Google map in the element of the ID map _ canvas. The watchPosition() method is called to measure the location of the device. The onSuccess callback method is called when the location of the device is successfully measured. In the onSuccess callback, the latitude and longitude of the device are displayed in the element of the ID geolocationinfo, along with the timestamp, that is, date and time of measuring the location of the device. The onError callback is set to display the error message if any error occurs while measuring the location of the device.

To display Google maps and to access the device location, we need to write permissions in the AndroidManifest.xml file as shown in Listing 8.7. Only the code in bold is new; the rest is the default code.

Listing 8.7 Code in the AndroidManifest.xml File

```
<?xml version = "1.0" encoding = "utf-8"?>
<manifest xmlns:android = "http://schemas.android.com/apk/res/android"
   package = "com.phonegap.pggooglemapapp"
   android:versionCode = "1"
   android:versionName = "1.0" >
   <uses-sdk
      android:minSdkVersion = "11"
      android:targetSdkVersion = "17"/>
   <uses-permission
      android:name = "android.permission.ACCESS_NETWORK_STATE"/>
      <uses-permission android:name = "android.permission.INTERNET"/>
      <uses-permission android:name = "android.permission.ACCESS_FINE_LOCATION"/>
      <uses-permission android:name = "android.permission.
      ACCESS_COARSE_LOCATION"/>
      <uses-permission android:name = "android.permission.
      ACCESS_LOCATION_EXTRA_COMMANDS"/>
   <application
      android:allowBackup = "true"
      android:icon = "@drawable/ic_launcher"
      android:label = "@string/app_name"
      android:theme = "@style/AppTheme" >
      <activity android:name = "com.phonegap.pggooglemapapp.
      PGGoogleMapAppActivity"
         android:label = "@string/app_name" >
         <intent-filter>
            <action android:name = "android.intent.action.MAIN"/>
```

```
            <category android:name = "android.intent.category.LAUNCHER"/>
        </intent-filter>
    </activity>
  </application>
</manifest>
```

Our application is ready to run. Start the `PhoneGapAVD` and run the application. Remember, your PC must be connected to the Internet. The Google map will appear in the emulator. To supply the location of the device, that is, to supply longitude and latitude values, we will make use of the Dalvik Debug Monitor Server (DDMS). To switch to the DDMS perspective, either click the DDMS icon in the toolbar or choose the `Window->Open Perspective->DDMS` option. In the DDMS perspective, we find our Android emulator, `PhoneGapAVD` running with ID `emulator-5554`. Select the emulator, `emulator-5554`, from the left pane and the `Emulator Control` button from the right pane. Under the `Location Controls` tab, we find two text boxes, `Longitude` and `Latitude`, that can be used for sending longitude and latitude values to the application. Let us supply the longitude and latitude values for the device as –122.43444 and 7.76944, respectively, followed by clicking the `Send` button, as shown in Figure 8.5.

Because the longitude and latitude values of San Francisco are supplied in the application above, the Google map will appear centered around San Francisco. Also, the geolocation information of the device that we supplied through the DDMS is displayed at the top (see Figure 8.6).

Displaying Google Map Centered at the Device Location

In the application above, the Google map was displayed centered at the supplied longitude and latitude values. Now, let us modify the application so that the Google map is displayed centered at the device's location. That is, the application should first detect the location of the device and use that information in centering the Google map. To do so, open the `PGGoogleMapApp` that we created above. Open the `index.html` file in the `assets/www` folder and modify it to appear as shown in Listing 8.8. Observe the code in bold; the rest of the code is the same as we saw in Listing 8.6.

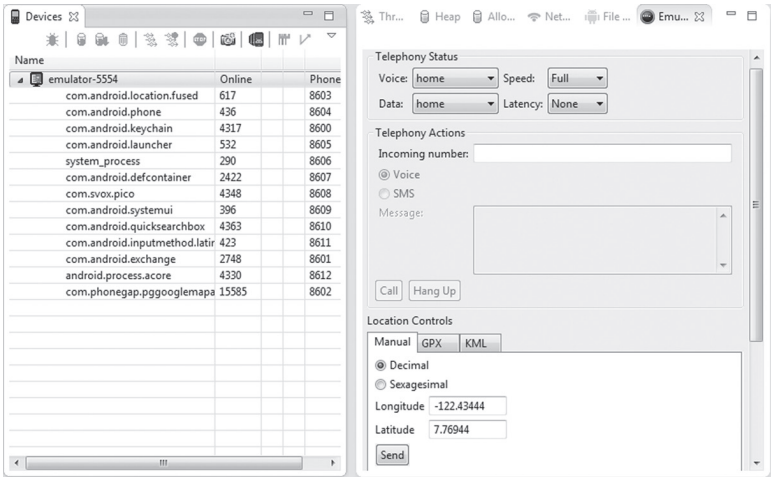


Figure 8.5 Supplying longitude and latitude values of the device through the DDMS perspective.

Figure 8.6 A Google map is displayed centered at the specified location, along with the geolocation information displayed at the top.

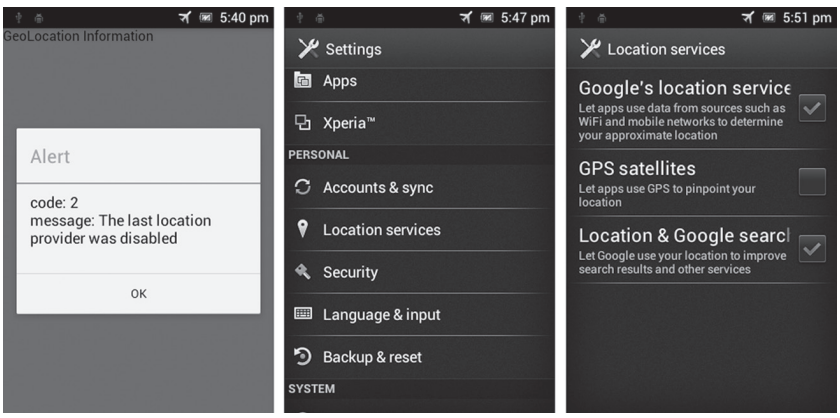
Listing 8.8 Code Written in the `index.html` File

```
<!DOCTYPE HTML>
<html>
   <head>
   <title>PhoneGap Application</title>
   <script type = "text/javascript" charset = "utf-8" src = "cordova-2.3.0.js">
   </script>
   <script type = "text/javascript" src = "http://maps.google.com/maps/api/
   js?sensor = false"></script>
   <link href = "http://code.google.com/apis/maps/documentation/javascript/
   examples/default.css" rel = "stylesheet" type = "text/css"/>
   <script type = "text/javascript">
   function onBodyLoad() {
      document.addEventListener("deviceready", PhonegapLoaded, false);
   }
   function PhonegapLoaded(){
      navigator.geolocation.getCurrentPosition(onSuccess, onError,
      {enableHighAccuracy: true});
   }
   function onSuccess(position) {
      var element = document.getElementById('geolocationinfo');
      element.innerHTML = 'Latitude: ' + position.coords.latitude + '<br/>' +
      'Longitude: ' + position.coords.longitude + '<br/>' +
      'Timestamp: ' + new Date(position.timestamp) + '<br/>';
      var mapOptions = {
```

```
            center: new google.maps.LatLng(position.coords.latitude, position.coords.
            longitude),
            zoom: 8,
            mapTypeId: google.maps.MapTypeId.ROADMAP
         };
         var map = new google.maps.Map(document.getElementById("map_canvas"),
         mapOptions);
      }
      function onError(error) {
         alert('code: ' + error.code + '\n' + 'message: ' + error.message + '\n');
      }
      </script>
      </head>
      <body onload = "onBodyLoad()">
         <div id = "geolocationinfo">GeoLocation Information</div>
         <div id = "map_canvas" style = "width:100%; height:100%"></div>
      </body>
</html>
```

We can see that after the PhoneGap is loaded, the `getCurrentPosition()` method is called to find the device location. Upon measuring the device location, the `onSuccess` callback is called and the `position` object containing the geolocation information of the device is passed to it. In the `onSuccess` callback, the longitude and latitude of the device are displayed through the `<div>` element of the ID `geolocationinfo`. Also, the timestamp where the device location is measured is displayed. Thereafter, the `mapOptions` object is defined to center the Google map at the device location's latitude and longitude values. Finally, an instance of `Map` class is created called `map` to display the Google map in the `<div>` element of the ID `map _ canvas`.

If we run the application on the Android device, we might get an error, as shown in Figure 8.7a. The error occurs if `Location services` are not enabled on the device. To enable `Location services` on a device, open the `Settings` menu on it and select the `Location services` option (see Figure 8.7b) in it. From the `Location services` menu, enable both



(a) (b) (c)

Figure 8.7 (a) An error code is displayed if `Location` services are not enabled. (b) The setting menu on the device showing different options. (c) The location services menu showing different options.

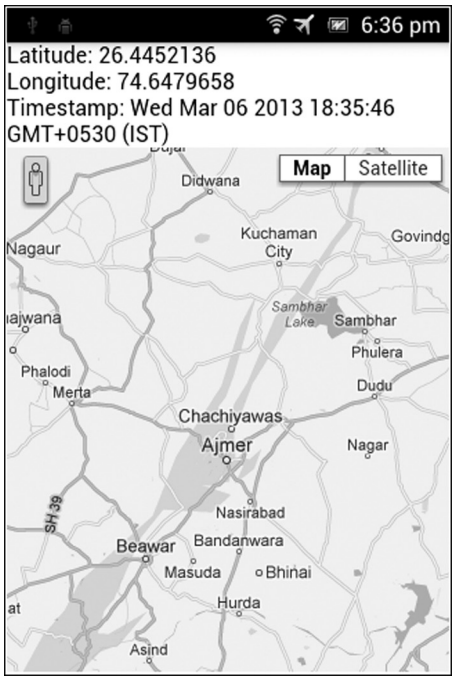


Figure 8.8 A Google map is displayed centered at the device location.

the `Google's location service` and `Location & Google search` options (see Figure 8.7c). Also, do not forget to switch on the WiFi on the device.

After enabling the `Location services` on the device, when we run the application, the Google map will appear centered at the device location, as shown in Figure 8.8. Also, the geolocation information of the device in terms of latitude and longitude is displayed along with the timestamp above the map.

Displaying Markers on the Map

To display markers at the specified location coordinates on the map, the `google.maps.Marker` class is used. The instance of `google.maps.Marker` class is created and options are defined to indicate the location of the marker. Given below are the four options that are popularly used while creating markers:

- `position`—This is a required option and is used to indicate the location of the marker on the map. An instance of `google.maps.LatLng` class is created passing the desired latitude and longitude values to define the location of the marker.
- `title`—This is an optional option that is used to display text that appears hovering over the marker.
- `draggable`—A Boolean value that determines whether the marker can be dragged or not. The Boolean value `true`, if assigned to this option, will make the marker draggable.

- `raiseOnDrag`—Boolean value when set to `true` makes the marker rise when dragged and lowers it when dropped.

To display markers in the Google map that we displayed above, open the `PGGoogleMapApp` and modify its `index.html` file to appear as shown in Listing 8.9. Only the code in bold is modified; the rest of the code is the same as we saw in Listing 8.8.

Listing 8.9 Code Written in the `index.html` File

```
<!DOCTYPE HTML>
<html>
   <head>
   <title>PhoneGap Application</title>
   <script type = "text/javascript" charset = "utf-8" src = "cordova-2.3.0.js">
   </script>
   <script type = "text/javascript" src = "http://maps.google.com/maps/api/
   js?sensor = false"></script>
   <link href = "http://code.google.com/apis/maps/documentation/javascript/
   examples/default.css" rel = "stylesheet" type = "text/css"/>
   <script type = "text/javascript">
   function onBodyLoad() {
      document.addEventListener("deviceready", PhonegapLoaded, false);
   }
   function PhonegapLoaded(){
      navigator.geolocation.getCurrentPosition(onSuccess, onError,
      {enableHighAccuracy: true});
   }
   function onSuccess(position) {
      var element = document.getElementById('geolocationinfo');
      element.innerHTML = 'Latitude: ' + position.coords.latitude + '<br/>' +
      'Longitude: ' + position.coords.longitude + '<br/>' +
      'Timestamp: ' + new Date(position.timestamp) + '<br/>';
      var mapOptions = {
         center: new google.maps.LatLng(position.coords.latitude, position.coords.
         longitude),
         zoom: 8,
         mapTypeId: google.maps.MapTypeId.ROADMAP
      };
      var map = new google.maps.Map(document.getElementById("map_canvas"),
      mapOptions);
      var marker = new google.maps.Marker({
         position: new google.maps.LatLng(position.coords.latitude, position.
         coords.longitude),
         title: "I am here"
         });
         marker.setMap(map);
   }
   function onError(error) {
      alert('code: ' + error.code + '\n' + 'message: ' + error.message + '\n');
   }
   </script>
   </head>
   <body onload = "onBodyLoad()">
      <div id = "geolocationinfo">GeoLocation Information</div>
      <div id = "map_canvas" style = "width:100%; height:100%"></div>
   </body>
</html>
```

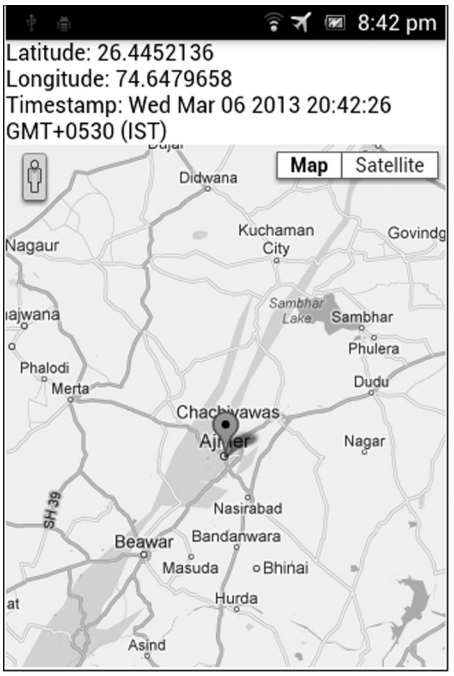


Figure 8.9 A Google map is displayed centered at the device location and a marker is displayed at the device location.

In the code above, we see that an instance of `Marker` class is created by the name `marker` and its `position` property is set equal to the longitude and latitude values of the device. The `title` of the marker is set to the text `I am here`. Upon running the application, the Google map will be displayed centered at the device location and the device location will be identified by a marker, as shown in Figure 8.9.

Drawing Lines on the Map

To draw lines on the map, the `Polyline` class is used. The `Polyline` class supports lots of properties to set the color, opacity, path, map, and so forth, of the line. The following statements draw a red line between two positions, `p1` and `p2`:

```
var line = new google.maps.Polyline({
   p1: new google.maps.LatLng(latitude_value1, longitude_value1),
   p2: new google.maps.LatLng(latitude_value2, longitude_value2),
   path: [p1, p2],
   strokeColor: '#FF0000',
   strokeOpacity: 1.0,
   map: map
});
```

We can see that a line is drawn between two positions, `p1` and `p2`, where `p1` and `p2` are the `LatLng` instances defined by supplying the respective latitude and longitude values. Similarly, the following statements draw a red line between two markers, `marker1` and `marker2`:

```
var line = new google.maps.Polyline({
   path: [marker1.getPosition(), marker2.getPosition()],
   strokeColor: '#FF0000',
   strokeOpacity: 1.0,
   map: map
});
```

Note: The map property is essential in `Polyline` and other drawing classes because it defines the map object on which the line has to be drawn.

Finding the Distance between Two Positions

To find out the distance between two position objects, the `computeDistanceBetween()` method is used, as shown in the statements below:

```
var p1 = new google.maps.LatLng(latitude_value1, longitude_value1);
var p2 = new google.maps.LatLng(latitude_value2, longitude_value2);
d = google.maps.geometry.spherical.computeDistanceBetween(p1, p2);
```

We can see that positions, that is, `LatLng` instances, `p1` and `p2` are defined by supplying the respective latitude and longitude values. The distance between the two positions is computed by the `computeDistanceBetween()` method and will be assigned to variable `d`. The distance computed will be in meters.

Let us modify the application `PGGoogleMapApp` to display a line between two markers in the Google map and display the distance between the two markers. We will be displaying the two markers at the following two locations:

- The first marker will be displayed at the device's location.
- The second marker will be displayed at the location near the device location. If the device location (or the first marker) is represented by `latitude` and `longitude` values, then the location of the second marker is computed by adding `.5` to the `latitude` and `longitude`. That is, the second marker will be displayed at a location represented by `latitude+.5` and `longitude+.5` values.

So, open the `index.html` file in the `PGGoogleMapApp` application and modify it to appear as shown in Listing 8.10. Only the code in bold is modified; the rest of the code is the same as we saw in Listing 8.9.

Listing 8.10 Code Written in the `index.html` File

```
<!DOCTYPE HTML>
<html>
   <head>
   <title>PhoneGap Application</title>
   <script type = "text/javascript" charset = "utf-8" src = "cordova-2.3.0.js">
   </script>
   <script type = "text/javascript" src = "http://maps.google.com/maps/api/
   js?sensor = false"></script>
```

```
<link href = "http://code.google.com/apis/maps/documentation/javascript/
examples/default.css" rel = "stylesheet" type = "text/css"/>
<script type = "text/javascript" src = "http://maps.google.com/maps/api/
js?sensor = false&libraries = geometry"> </script>
<script type = "text/javascript">
function onBodyLoad() {
   document.addEventListener("deviceready", PhonegapLoaded, false);
}
function PhonegapLoaded(){
   navigator.geolocation.getCurrentPosition(onSuccess, onError,
   {enableHighAccuracy: true});
}
function onSuccess(position) {
   var geolocationinfo1 = document.getElementById('geolocationinfo1');
   geolocationinfo1.innerHTML = 'Location 1: (Latitude: ' + position.coords.
   latitude + '<br/>' +
   'Longitude: ' + position.coords.longitude + '<br/>';
   var geolocationinfo2 = document.getElementById('geolocationinfo2');
   geolocationinfo2.innerHTML = 'Location 2: (Latitude: ' + position.coords.
   latitude +.5 + '<br/>' +
   'Longitude: ' + position.coords.longitude +.5+ '<br/>';
   var mapOptions = {
      center: new google.maps.LatLng(position.coords.latitude, position.coords.
      longitude),
      zoom: 8,
      mapTypeId: google.maps.MapTypeId.ROADMAP
   };
   var map = new google.maps.Map(document.getElementById("map_canvas"),
   mapOptions);
   var marker1 = new google.maps.Marker({
      position: new google.maps.LatLng(position.coords.latitude, position.
      coords.longitude),
      draggable: true,
      raiseOnDrag: false
   });
   marker1.setMap(map);
   var marker2 = new google.maps.Marker({
      position: new google.maps.LatLng(position.coords.latitude+.5, position.
      coords.longitude+.5),
      draggable: true,
      raiseOnDrag: false
   });
   marker2.setMap(map);
   var line = new google.maps.Polyline({
      path: [marker1.getPosition(), marker2.getPosition()],
      map: map
   });
   var p1 = new google.maps.LatLng(position.coords.latitude, position.coords.
   longitude);
   var p2 = new google.maps.LatLng(position.coords.latitude +.5, position.
   coords.longitude +.5);
   d = (google.maps.geometry.spherical.computeDistanceBetween(p1, p2)/1000).
   toFixed(2);
   document.getElementById('distance').innerHTML = "Distance between
   locations:" + d +" km";
}
function onError(error) {
   alert('code: ' + error.code + '\n' + 'message: ' + error.message + '\n');
}
</script>
</head>
```

```
    <body onload = "onBodyLoad()">
        <div id = "geolocationinfo1">GeoLocation Information 1</div>
        <div id = "geolocationinfo2">GeoLocation Information 2</div>
        <div id = "distance">Distance</div>
        <div id = "map_canvas" style = "width:100%; height:100%"></div>
    </body>
</html>
```

We can see that the `geometry` library is included in the application to compute the distance between two positions. Also, two `<div>` elements with IDs `geolocationinfo1` and `geolocationinfo2` are defined, where `geolocationinfo1` will be used to display the location information of the device and `geolocationinfo2` will be used to display the location information near the device. The `<div>` element with the ID `distance` will be used for displaying the distance between the two locations in kilometers. A `mapOptions` object is defined that centers the Google map at the device location. An instance of the `Map` class is created called `map` to display the Google map. Two markers, `marker1` and `marker2`, are displayed on the Google map. `marker1` is displayed at the device location, and `marker2` is displayed near the device location. An instance of `Polyline` is created to draw a line between the two markers. Two position objects, `p1` and `p2`, are defined where position object `p1` points at the latitude and longitude of the device and position object `p2` points at the location near the device (latitude + .5 and longitude + .5). Using the `computeDistanceBetween()` method, the distance between two position objects `p1` and `p2` is computed and is displayed through the `<div>` element, `distance`.

Upon running the application, two markers will be displayed on the Google map, one at the device location and the other near it. Also, a line is drawn between the two markers. In addition, the distance between the two markers is also computed and displayed (see Figure 8.10).

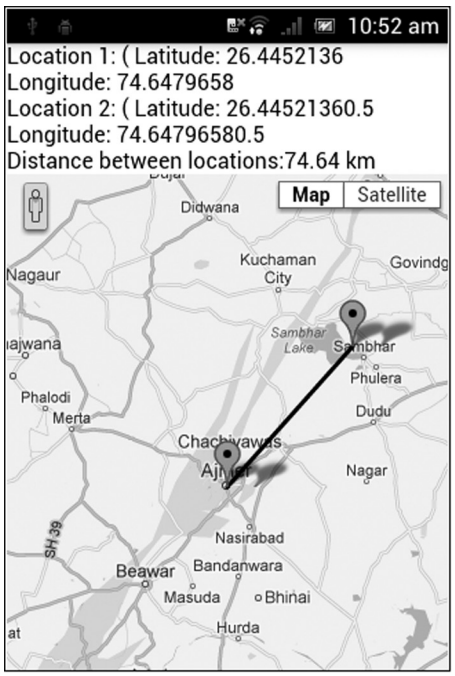


Figure 8.10 A line is drawn between the two markers and the distance between them is computed.

Dragging Markers and Computing Distances

In the application above, we saw that the two markers are disabled; that is, we cannot move or drag them. Let us modify the application above so that we can drag either of the two markers. Also, when we drag either of the markers, their new geolocation information will be displayed and the new distance between the two markers will be computed and displayed.

So, open the application `PGGoogleMapApp` and modify the `index.html` file found in the `assets/www` folder to appear as shown in Listing 8.11. Only the code in bold is modified; the rest of the code is the same as we saw in Listing 8.10.

Listing 8.11 Code Written in the `index.html` File

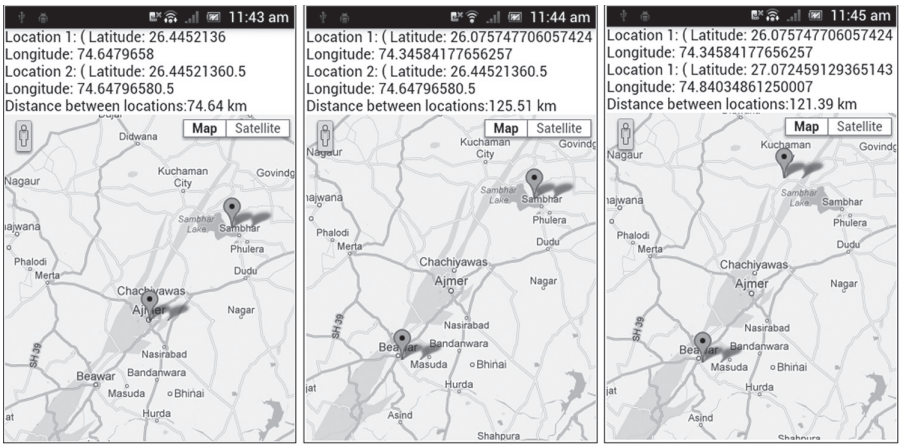
```
<!DOCTYPE HTML>
<html>
   <head>
   <title>PhoneGap Application</title>
   <script type = "text/javascript" charset = "utf-8" src = "cordova-2.3.0.js">
   </script>
   <script type = "text/javascript" src = "http://maps.google.com/maps/api/
   js?sensor = false"></script>
   <link href = "http://code.google.com/apis/maps/documentation/javascript/
   examples/default.css" rel = "stylesheet" type = "text/css"/>
   <script type = "text/javascript" src = "http://maps.google.com/maps/api/
   js?sensor = false&libraries = geometry"></script>
   <script type = "text/javascript">
   function onBodyLoad() {
      document.addEventListener("deviceready", PhonegapLoaded, false);
   }
   function PhonegapLoaded(){
      navigator.geolocation.getCurrentPosition(onSuccess, onError,
      {enableHighAccuracy: true});
   }
   function onSuccess(position) {
      var geolocationinfo1 = document.getElementById('geolocationinfo1');
      geolocationinfo1.innerHTML = 'Location 1: (Latitude: ' + position.coords.
      latitude + '<br/>' +
      'Longitude: ' + position.coords.longitude + '<br/>';
      var geolocationinfo2 = document.getElementById('geolocationinfo2');
      geolocationinfo2.innerHTML = 'Location 2: (Latitude: ' + position.coords.
      latitude +.5 + '<br/>' +
      'Longitude: ' + position.coords.longitude +.5+ '<br/>';
      var mapOptions = {
         center: new google.maps.LatLng(position.coords.latitude, position.coords.
         longitude),
         zoom: 8,
         mapTypeId: google.maps.MapTypeId.ROADMAP
      };
      var map = new google.maps.Map(document.getElementById("map_canvas"),
      mapOptions);
      var marker1 = new google.maps.Marker({
         position: new google.maps.LatLng(position.coords.latitude, position.
         coords.longitude),
         draggable: true,
         raiseOnDrag: false
      });
      marker1.setMap(map);
      var marker2 = new google.maps.Marker({
```

```
            position: new google.maps.LatLng(position.coords.latitude+.5, position.
            coords.longitude+.5),
            draggable: true,
            raiseOnDrag: false
         });
         marker2.setMap(map);
         findDistance(marker1, marker2);
         google.maps.event.addListener(marker1, 'drag', function() {
            var geolocationinfo1 = document.getElementById('geolocationinfo1');
            geolocationinfo1.innerHTML = 'Location 1: (Latitude: ' + marker1.
            getPosition().lat() + '<br/>' +
            'Longitude: ' + marker1.getPosition().lng() + '<br/>';
            findDistance(marker1, marker2);
         });
         google.maps.event.addListener(marker2, 'drag', function() {
            var geolocationinfo2 = document.getElementById('geolocationinfo2');
            geolocationinfo2.innerHTML = 'Location 1: (Latitude: ' + marker2.
            getPosition().lat() + '<br/>' +
            'Longitude: ' + marker2.getPosition().lng() + '<br/>';
            findDistance(marker1, marker2);
         });
      }
      function findDistance(marker1, marker2){
         var p1 = new google.maps.LatLng(marker1.getPosition().lat(), marker1.
         getPosition().lng());
         var p2 = new google.maps.LatLng(marker2.getPosition().lat(), marker2.
         getPosition().lng());
         d = (google.maps.geometry.spherical.computeDistanceBetween(p1, p2)/1000).
         toFixed(2);
         document.getElementById('distance').innerHTML = "Distance between
         locations:" + d +" km";
      }
      function onError(error) {
         alert('code: ' + error.code + '\n' + 'message: ' + error.message + '\n');
      }
      </script>
      </head>
      <body onload = "onBodyLoad()">
         <div id = "geolocationinfo1">GeoLocation Information 1</div>
         <div id = "geolocationinfo2">GeoLocation Information 2</div>
         <div id = "distance">Distance</div>
         <div id = "map_canvas" style = "width:100%; height:100%"></div>
      </body>
</html>
```

We can see that the function `findDistance()` is defined and uses two parameters, `marker1` and `marker2`. Using the parameters `marker1` and `marker2`, the `findDistance()` function retrieves the position objects, `p1` and `p2`, that represent the positions of the two markers. Thereafter, the `computeDistanceBetween()` method is used to compute the distance between the two position objects, `p1` and `p2`. The computed distance is converted into kilometers and displayed through the `<div>` element, `distance`.

Also, the `drag` event listeners are associated with the two markers so that whenever either of them is dragged, their new geolocation information is displayed through their respective `<div>` element and the `findDistance()` function is called to recompute the distance between the two markers.

Upon running the application, we see that the Google map is displayed with two markers. One marker will be displayed at the device's current location, and the other marker will be displayed



(a) (b) (c)

Figure 8.11 (a) A Google map is displayed with two markers. (b) The geolocation information of `marker1` and updated distance. (c) The geolocation information of `marker2` along with the updated distance.

near the device location. The geolocation information of the two markers will be displayed above the map. Also, the distance between the two markers will be computed and displayed in kilometers (see Figure 8.11a). Upon dragging the first marker on the Google map, that is, `marker1`, its geolocation information displayed above the map will change to reflect its current location. Also, the current distance between the two markers will be computed and displayed as shown in Figure 8.11b. Similarly, upon dragging the second marker, again, its geolocation information above the map will be updated along with the distance between the two markers (see Figure 8.11c).

Until now, we have been displaying the geolocation of the device in terms of latitude and longitude values. How about displaying addresses of the locations?

Using the Geocoder Class

The `Geocoder` class is used to convert an address to a `LatLng` object, and vice versa. It includes a method, `geocode()`, that takes a `GeocoderRequest` object as a parameter, and a callback function that can be used to retrieve the device's complete address. The statements below show how the given latitude and longitude values can be converted into a formatted address using the `Geocoder` class:

```
var latlng = new google.maps.LatLng(37.76944, -122.43444);
var geocoder = new google.maps.Geocoder();
geocoder.geocode({'latLng': latlng}, function(resp, status) {
   if (resp[0]) {
      alert("Formatted address is "+ resp[0].formatted_address);
   }
}
```

We can see that a `LatLng` object called `latlng` is created, passing the specified longitude and latitude values. After creating the `latlng` object, the `geocode()` method is called on the

`Geocoder` instance, `geocoder`. If no error occurs while executing the `geocode()` method, the `resp` parameter in the callback function will contain the formatted address that represents the supplied latitude and longitude values. The alert dialog uses the first element in the `resp` array to display the formatted address.

Let us modify the application above so that instead of the latitude and longitude values, the formatted addresses of the markers are displayed. Also, when we drag either of the markers on the Google map, the formatted address must change to reflect the new location. In addition, whenever either of the two markers is dragged, we want the new distance between the two markers to be computed and displayed.

So, open the application `PGGoogleMapApp` and modify the `index.html` file found in the `assets/www` folder to appear as shown in Listing 8.12. Only the code in bold is modified; the rest of the code is the same as we saw in Listing 8.11.

Listing 8.12 Code Written in the `index.html` File

```
<!DOCTYPE HTML>
<html>
   <head>
   <title>PhoneGap Application</title>
   <script type = "text/javascript" charset = "utf-8" src = "cordova-2.3.0.js">
   </script>
   <script type = "text/javascript" src = "http://maps.google.com/maps/api/
   js?sensor = false"></script>
   <link href = "http://code.google.com/apis/maps/documentation/javascript/
   examples/default.css" rel = "stylesheet" type = "text/css"/>
   <script type = "text/javascript" src = "http://maps.google.com/maps/api/
   js?sensor = false&libraries = geometry"></script>
   <script type = "text/javascript">
   function onBodyLoad() {
      document.addEventListener("deviceready", PhonegapLoaded, false);
   }
   function PhonegapLoaded(){
      navigator.geolocation.getCurrentPosition(onSuccess, onError,
      {enableHighAccuracy: true});
   }
   function onSuccess(position) {
      var lat = parseFloat(position.coords.latitude);
      var lng = parseFloat(position.coords.longitude);
      var latlng = new google.maps.LatLng(lat, lng);
      var geocoder = new google.maps.Geocoder();
      geocoder.geocode({'latLng': latlng}, function(resp, status) {
         if (resp[0]) {
            var geolocationinfo1 = document.getElementById('geolocationinfo1');
            geolocationinfo1.innerHTML = 'Location 1: '+ resp[0].
            formatted_address;
         }
      });
      var lat2 = parseFloat(position.coords.latitude+.5);
      var lng2 = parseFloat(position.coords.longitude+.5);
      var latlng2 = new google.maps.LatLng(lat2, lng2);
      geocoder.geocode({'latLng': latlng2}, function(resp, status) {
         if (resp[0]) {
            var geolocationinfo2 = document.getElementById('geolocationinfo2');
            geolocationinfo2.innerHTML = 'Location 2: '+ resp[0].
            formatted_address;
         }
```

```
    });
    var mapOptions = {
       center: new google.maps.LatLng(position.coords.latitude, position.coords.
       longitude),
       zoom: 8,
       mapTypeId: google.maps.MapTypeId.ROADMAP
    };
    var map = new google.maps.Map(document.getElementById("map_canvas"),
    mapOptions);
    var marker1 = new google.maps.Marker({
       position: new google.maps.LatLng(position.coords.latitude, position.
       coords.longitude),
       draggable: true,
       raiseOnDrag: false
    });
    marker1.setMap(map);
    var marker2 = new google.maps.Marker({
       position: new google.maps.LatLng(position.coords.latitude+.5, position.
       coords.longitude+.5),
       draggable: true,
       raiseOnDrag: false
    });
    marker2.setMap(map);
    findDistance(marker1, marker2);
    google.maps.event.addListener(marker1, 'drag', function() {
       var geocoder = new google.maps.Geocoder();
       geocoder.geocode({'latLng': marker1.getPosition()}, function(resp,
       status) {
          if (resp[0]) {
             var geolocationinfo1 = document.getElementById('geolocationinfo1');
             geolocationinfo1.innerHTML = 'Location 1: '+ resp[0].
             formatted_address;
          }
       });
       findDistance(marker1, marker2);
    });
    google.maps.event.addListener(marker2, 'drag', function() {
       var geocoder = new google.maps.Geocoder();
       geocoder.geocode({'latLng': marker2.getPosition()}, function(resp,
       status) {
          if (resp[0]) {
             var geolocationinfo2 = document.getElementById('geolocationinfo2');
             geolocationinfo2.innerHTML = 'Location 2: '+ resp[0].
             formatted_address;
          }
       });
       findDistance(marker1, marker2);
    });
}
function findDistance(marker1, marker2) {
    var p1 = new google.maps.LatLng(marker1.getPosition().lat(), marker1.
    getPosition().lng());
    var p2 = new google.maps.LatLng(marker2.getPosition().lat(), marker2.
    getPosition().lng());
    d = (google.maps.geometry.spherical.computeDistanceBetween(p1, p2)/1000).
    toFixed(2);
    document.getElementById('distance').innerHTML = "Distance between
    locations:" + d +" km";
}
function onError(error) {
    alert('code: ' + error.code + '\n' + 'message: ' + error.message + '\n');
```

```
    }
    </script>
    </head>
    <body onload = "onBodyLoad()">
       <div id = "geolocationinfo1">GeoLocation Information 1</div>
       <div id = "geolocationinfo2">GeoLocation Information 2</div>
       <div id = "distance">Distance</div>
       <div id = "map_canvas" style = "width:100%; height:100%"></div>
    </body>
</html>
```

In the code above, we see that in the `onSuccess` callback method that is called on successful execution of the `getCurrentPosition()` method, the latitude and longitude values of the current device location are retrieved. Using the retrieved latitude and longitude values, the `latlng` instance of the `LatLng` class is defined. The `latlng` instance is passed to the `geocode()` method of the `Geocoder` instance to convert the latitude and longitude values into the formatted address. The formatted address of the device is then displayed on the screen through the `<div>` element of the ID `geolocationinfo1`. Similarly, the location near the device is converted into a formatted address and displayed through the `<div>` element of the ID `geolocationinfo2`. The distance between the two markers is computed and displayed through the `findDistance()` function.

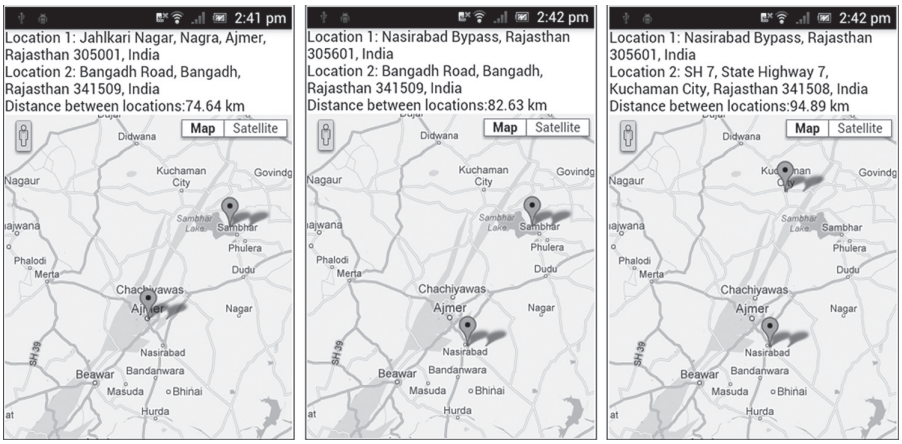
The `drag` event listeners are associated with the two markers `marker1` and `marker2`. Whenever either of the markers is dragged, the `geocode()` method on the `Geocoder` class is called, supplying the new marker position to retrieve the formatted address representing the new marker location. The new formatted address is then displayed through the respective `<div>` elements. Each time either of the markers is dragged, the `findDistance()` function is also called to compute the new distance between the markers.

Upon running the application, we see that the Google map is displayed with two markers. One marker will be displayed at the device current location, and the other marker will be displayed near the device location. The geolocation information of the two markers will be displayed as a formatted address at the top of the map. Also, the distance between the two markers will be computed and displayed in kilometers (see Figure 8.12a). On dragging the first marker, that is, `marker1` on the Google map, its formatted address will be changed to represent the new marker location. Also, the current distance between the two markers will be computed and displayed as shown in Figure 8.12b. Similarly, upon dragging the second marker, its formatted address will be modified to represent the new location of `marker2`. Again, the distance between the two markers will be updated (see Figure 8.12c).

Using Compass API

The compass indicates the direction in which the device is pointing. It provides the device's heading in degrees from true north in a clockwise direction. Using the PhoneGap Compass API, when we turn the device in any direction, a number between 0 and 359.99 is displayed that corresponds to the direction in which the device is pointing. A value of 0 indicates the device is pointing north, 90 indicates it is pointing east, 180 refers to south, and 270 refers to west.

Note: Not all smartphones have a compass.



(a) (b) (c)

Figure 8.12 (a) A Google map is displayed with two markers and the formatted addresses of the markers are displayed at the top. (b) The new formatted address of `marker1` and the distance is updated. (c) The new formatted address of `marker2` along with the updated distance.

To know the direction in which the device is heading or pointing, the `getCurrentHeading()` method is used as shown below:

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

The `onSuccess` callback method is called when the direction in which the device is pointing is successfully measured. The `onError` callback method is called when some error occurs while reading the compass. With the `onSuccess` callback, a `heading` object is passed as a parameter. The properties in the `heading` object can be used to display detailed information about the device's orientation. Below are the properties supported by the `heading` object:

- `magneticHeading`—Represents the device's current heading in degrees ranging from 0 to 359.99.
- `trueHeading`—Represents the device's current heading relative to the geographic North Pole in degrees ranging from 0 to 359.99. There are two North Poles; one is the geographic North Pole and the other is the magnetic North Pole, which keeps moving because of magnetic changes in the earth's core.
- `headingAccuracy`—Represents the deviation between the `magneticHeading` and `trueHeading` values in degrees.
- `timestamp`—Represents the time in milliseconds when the heading is measured since January 1, 1970.

In order to keep watching or monitoring the device heading at regular intervals, the `watchHeading()` method is called as shown below:

```
navigator.compass.watchHeading (onSuccess, onError,compassOptions);
```

The `compassOptions` object is used for defining the period or time interval in milliseconds to update the heading. For example, the code given below measures the direction in which the device is pointing every 3 milliseconds (3 seconds):

```
var compassOptions = {frequency: 3000};
var watchID = navigator.compass.watchHeading(onSuccess, onError,
compassOptions);
```

Note: If the `compassOptions` object is not provided, the default interval of 1000 milliseconds is used.

To stop measuring the device heading, the `clearwatch()` method is called and the reference of the `watchID` that we want to stop or cancel is passed to it as shown below:

```
navigator.compass.clearWatch(watchID);
```

Let us apply the knowledge gained so far to finding the direction in which a device is heading or pointing. So, create an Android project called `PGCompassApp` and configure it for using PhoneGap. In the `assets/www` folder, open the `index.html` file and write the code as shown in Listing 8.13.

Listing 8.13 Code Written in the `index.html` File

```
<!DOCTYPE HTML>
<html>
   <head>
   <title>PhoneGap Application</title>
   <script type = "text/javascript" charset = "utf-8" src = "cordova-2.3.0.js">
   </script>
   <script type = "text/javascript">
   function onBodyLoad() {
      document.addEventListener("deviceready", PhonegapLoaded, false);
   }
   function PhonegapLoaded(){
      navigator.compass.getCurrentHeading(onSuccess, onError);
   }
   function onSuccess(heading) {
   var date = new Date(heading.timestamp);
   alert("Timestamp: "+ date+"\n"+
   "Magnetic Heading: "+heading.magneticHeading+"\n"+
   "True Heading: "+heading.trueHeading+"\n"+
   "Heading Accuracy: "+heading.headingAccuracy+"\n");
}
   function onError(err) {
   if (err.code = = CompassError.COMPASS_NOT_SUPPORTED)
   alert("Compass not supported.");
   else if (err.code = = CompassError.COMPASS_INTERNAL_ERR)
   alert("Compass Internal Error");
   else
   alert("Unknown Error");
}
   </script>
   </head>
   <body onload = "onBodyLoad()">
   </body>
</html>
```

In the code above, we can see that after PhoneGap is loaded, the `getCurrentHeading()` method is called to find the direction in which the device is heading. If the direction of the device is measured successfully, the `onSuccess` callback method will be called. The properties supported by the heading object passed to the `onSuccess` callback are used to display the magnetic heading, true heading, and heading accuracy of the device. Also, the timestamp at which the heading is measured is also displayed. In case some error occurs, the `onError` callback will be called to display the reason that caused the error.

Summary

In this chapter, we learned about different Phone APIs. We learned how to find acceleration of the device along the three axes using the Accelerometer API. We learned how to capture images using the Camera API. We saw the procedure for using the Capture API in capturing audio and video. We also saw how the Geolocation API can be used to find the device location. We looked at the methods used to watch the device's location at regular intervals. We also saw how Google map is displayed and how it can be centered at the device's current location. Finally, we saw how the Compass API is used to find the direction in which the device is pointing.

In Chapter 9, we will learn about Sencha Touch. We will learn how to download Sencha Touch and how it can be integrated with PhoneGap.

Chapter 9

Sencha Touch

In this chapter, we cover:

- Introducing Sencha Touch
- Model View Controller (MVC) architecture
- Enabling Web servers
- Installing Sencha Touch 2 Software Development Kit (SDK)
- Using the Sencha Cmd
- Folder Organization of the Sencha Touch App
- Understanding the component class, container classes, and layouts
- Creating views
- Understanding the `Index.html` file, using the `Ext.application` class
- Sencha Touch fields
- Using `xtype`
- Event handling
- Getting components
- Creating a welcome application
- Welcome application in the MVC format
- Creating a login form
- Creating a list application
- Navigating using tabs

Introducing Sencha Touch

Sencha Touch is a powerful cross-platform framework that enables developers to develop HTML5-based mobile applications that are compatible with Android, iOS, and BlackBerry devices. Below are a few of the features of this mobile framework:

- Sencha Touch is built specifically for mobile devices. Developers can develop applications that appear like native applications.

- It is based on Web standards such as HTML5, CSS3, and JavaScript.
- It supports audio, video, animations, and local storage to develop dynamic, optimized, and fully featured applications.
- It enables the integration of GPS and device sensor data into applications.
- Excellent documentation makes it very easy to use.
- The Sencha Touch classes support the MVC architecture.

Overview of the Sencha Touch Class Library

Sencha Touch comes bundled with several classes and components that can be readily used in developing applications rapidly. The following are the families of classes that are included in the Sencha Touch 2 class library:

- **Model classes**—Used to describe the business entities that are managed by the application. These classes basically help in defining the data to operate on.
- **Communication classes**—Used to describe storage and communication techniques used in the application. These classes help to consume data from different resources.
- **Views classes**—Used to describe the user interface of an application.
- **Controller classes**—Used to describe the business logic of an application. It defines the views to be displayed when a certain action is performed on the models.
- **Foundation utility classes**—Describes ready-to-use code. We can use different methods defined in the utility classes to perform lots of tasks straight away without writing a single line of code.

Before we continue, let us briefly review the MVC architecture.

Model View Controller (MVC) Architecture

An application with all data and business logic in a single file makes it hard to not only manage but also debug. Such applications are also difficult to read. MVC architecture splits or organizes the application files into the following three parts based on their functionality:

- **Models**—All data and storage files are kept in this category.
- **Views**—User interface-related files are kept in this category.
- **Controllers**—Business logic files that handle the user interactions and determine the next views to be displayed are kept in this category.

Because long application files are split and categorized on the basis of their respective functions, the applications in MVC architecture are easy to manage and debug.

To develop and test Sencha Touch applications, we need a Web server to host and run our applications. Let us learn the procedure to enable Web servers on two major operating systems.

Enabling Web Servers

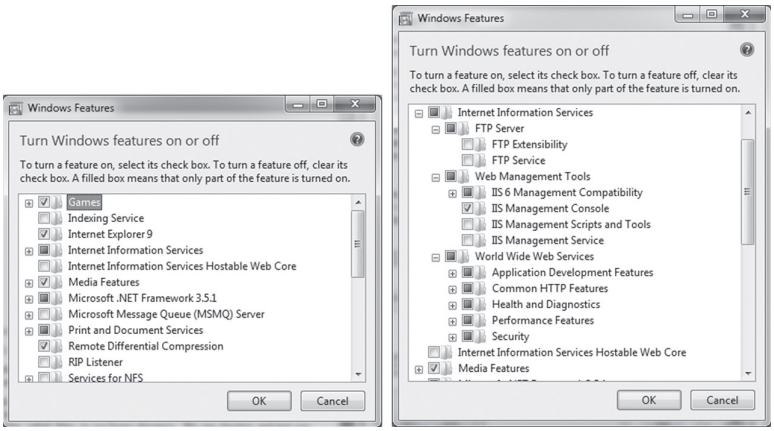
The procedures below are given to enable Web servers on Mac OSX and Windows operating systems:

- The Mac OSX already comes with a Web server installed. To enable it, open `System Preferences` and choose the `Sharing` option followed by enabling `Web Sharing`. Thereafter, click on the `Create Personal Website` folder to set up a Web folder called `Sites` in our home directory. We can develop and test our Sencha Touch applications in this `Sites` folder.
- Microsoft Windows comes with Internet Information Server (IIS) installed. To ensure that IIS is running, open the `Control Panel`, choose `Program and Features`, followed by selecting `Turn Windows features on or off` (in Vista or Windows 7). A window as shown in Figure 9.1a will open up. Upon expanding the `Internet Information Services` node in the `Windows Features` dialog, additional categories of IIS features will be displayed (see Figure 9.1b). Select `Internet Information Services` to choose the default features for installation. We can always enable additional features, like `Web Management Tools` or `World Wide Web Services`, if desired. After checking the checkboxes of the IIS features that we want to enable, click the `OK` button to initiate the procedure to enable IIS.

To confirm that IIS is successfully enabled on our machine, open the browser and type the following URL: `http://localhost`. If we get the output shown in Figure 9.2, it means the IIS is successfully enabled on our machine. The output displayed in Figure 9.2 appears on execution of the default `index.html` file that exists in the Web folder.

To manage and configure IIS, open `IIS Manager` using either of the following ways:

- Open the `Control Panel`, select `Administrative Tools`, and double-click the Internet Information Services (IIS) Manager shortcut.



(a) (b)

Figure 9.1 (a) The Windows `Features` dialog which will enable/disable different features. (b) The available features in the Internet Information Services node.

Figure 9.2 The output displayed upon execution of the default `index.html` file in the Web folder.

- Click `Start`, type `inetmgr` in the `Search programs and files` box, followed by pressing the `Enter` key.

By default, IIS does not serve the Multipurpose Internet Mail Extension (MIME) type files. So, we need to add a MIME type to IIS to enable it to serve MIME type files. To do so, open the properties for the server in IIS Manager, click `MIME Types` (see Figure 9.3), and click the `New` button.

In the `File` name extension box, type `JSON` and `application/json` in the `MIME type` box, followed by clicking the `OK` button (see Figure 9.4).

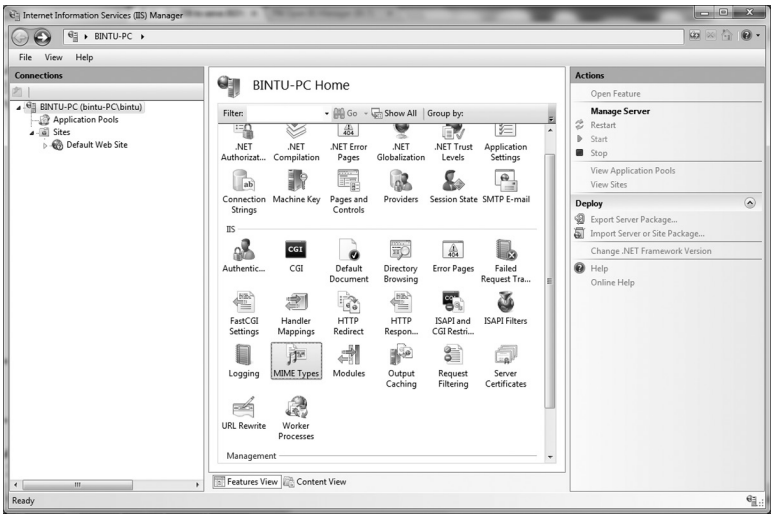


Figure 9.3 Windows displaying IIS properties.

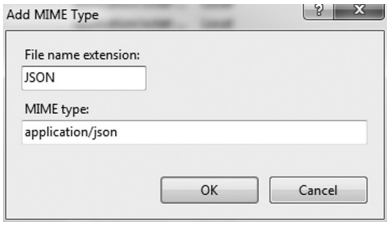


Figure 9.4 The dialog for adding MIME types to IIS.

Installing Sencha Touch 2 SDK

The steps below are given to install Sencha Touch 2 SDK:

- To download the Sencha Touch framework, visit the `http://www.sencha.com/products/touch/` URL and click the `Download` button. A zip file will be downloaded to our computer. The version of Sencha Touch available at the time of this writing is 2.1.1. Unzip the downloaded file and copy the directory containing the unzipped files to our Web folder. Open a Web browser and point at the directory containing the unzipped Sencha Touch files. For example, if the directory name in which the Sencha package is unzipped is `sencha-touch-2.1.1`, then point the browser to the `http://localhost/sencha-touch-2.1.1` address. The Sencha Touch demo page will open, as shown in Figure 9.5, which confirms that Sencha Touch is successfully installed on our computer.

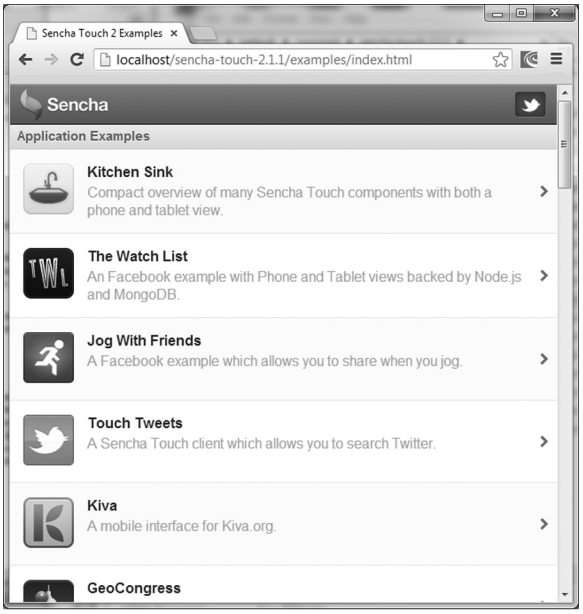


Figure 9.5 Sencha Touch demo page.

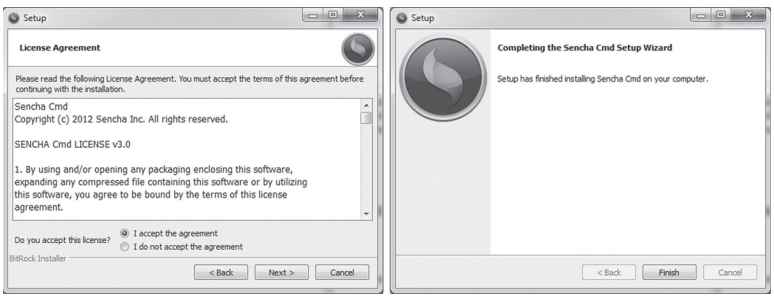
- The next step is to download the Sencha Touch Cmd from the URL `http://www.sencha.com/products/sencha-cmd/download`.
- Extract the SDK zip file to our Web folder, that is, in the `C:\inetpub\wwwroot` directory.
- To install the Sencha Touch Cmd installer, double-click the downloaded executable file of Sencha Cmd to invoke the `Sencha Cmd Setup Wizard`. The first screen is a welcome screen informing that Sencha Cmd utilities are used to package and deploy Sencha applications. Click the `Next` button to move further. The next dialog shows the license agreement and terms of using Sencha Cmd (see Figure 9.6a). Accept the agreement, followed by clicking the `Next` button to continue. The next dialog prompts us to specify the directory where we want to install Sencha Cmd. A default directory location will also be displayed. Keeping the default installation directory, click the `Next` button to continue. The next dialog informs us that the setup wizard is all set to install Sencha Cmd on our computer. Click the `Next` button to initiate the installation procedure. The Sencha Cmd files will be copied and installed on our computer. The final screen informs us that the Sencha Cmd has been successfully installed on our computer (see Figure 9.6b). Click the `Finish` button to exit the setup wizard. Sencha Cmd adds the Sencha command line tool to our path and enables us to generate new Sencha Touch application templates.

To confirm that Sencha Cmd is properly installed on our machine, change to the Sencha Touch directory, that is, `C:\inetpub\wwwroot\sencha-touch-2.1.1` directory, and type in the `sencha` command followed by the `Enter` key. If we get the output shown in Figure 9.7, it means that the Sencha Cmd has been correctly installed on our machine.

Using the Sencha Cmd

The Sencha Cmd command line tool performs several tasks. It adds the Sencha command line tool to our path and generates a new application template. Let us learn the procedure to create a Sencha Touch application using the Sencha Cmd tool.

Open a Terminal window and change the directory to the Sencha Touch folder that exists in the Web folder. Recall, we unzipped the Sencha package in a folder called `sencha-touch-2.1.1` into the Web folder `\inetpub\wwwroot`. In the Terminal window, we will give the `sencha`



(a) (b)

Figure 9.6 (a) The dialog displaying the License Agreement and terms of using Sencha Cmd. (b) The dialog confirming the successful installation of Sencha Cmd.

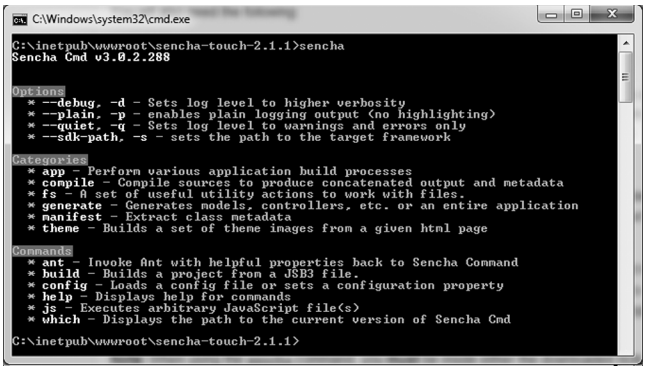


Figure 9.7 The output confirming that Sencha Cmd is correctly installed.

`generate app` command to create a new application. The `sencha generate app` command takes two parameters, the name of the application and its location, as shown below:

```
C:\inetpub\wwwroot\sencha-touch-2.1.1>sencha generate app MyApp.\MyApp
```

We can see that the name of our application is `MyApp` and the location defined for it is our Web folder. On giving the command above, the new application will be created and the Terminal window will show the output as shown in Figure 9.8.

The new application will be created and stored in the `MyApp` folder. Let us now look at different files and folders that are autogenerated for us.

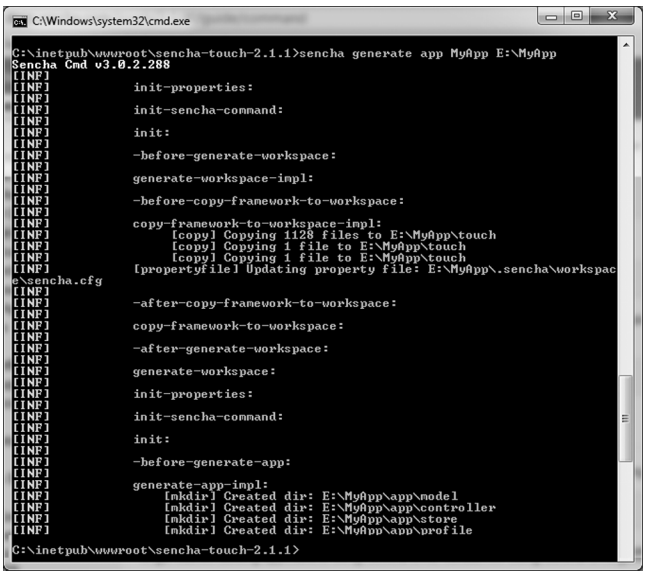


Figure 9.8 The output displayed upon generating a new application.

Folder Organization of the Sencha Touch App

Recall, the Sencha Touch classes support the MVC architecture, and hence the Sencha Touch applications have a well-defined folder structure that is enforced throughout the framework. Upon opening the `MyApp` folder, we see the list of files and folders that are automatically created for us as shown in Figure 9.9.

- **App**—Contains the code of our application. The location of the code can be changed through the `app.js` file. Inside the `app` folder, the application code is categorized into different folders as explained below.
- **Controller**—Contains all the classes used to interrelate the model and views. The files that accept user input, process the input, store the processed information in storage, load the views in response to the user interaction, and so on, are kept in this folder.
- **Model**—Contains the data model definitions. That is, the files that define the data and storage of the application are stored in this folder.
- **Profile**—Contains application profiles to support the user interface (UI) for different platforms. Sencha Touch supports three different profiles for phones, tablets, and desktop browsers. Hence, we can define separate UIs for different devices.
- **View**—Contains the files that define the user interface in the application.

Let us run the application to see the default output. Open the browser and type `http://localhost/MyApp`, followed by the `Enter` key. The application `MyApp` will run showing the output seen in Figure 9.10.

The output above is because of the default content in the `MyApp/app/view/Main.js` file. The default code in the `MyApp/app/view/Main.js` file is shown in Listing 9.1.

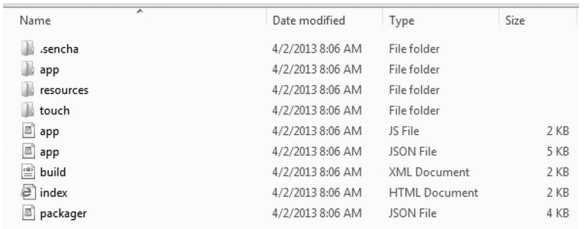


Figure 9.9 The list of files and folders that open in a typical Sencha Touch application.

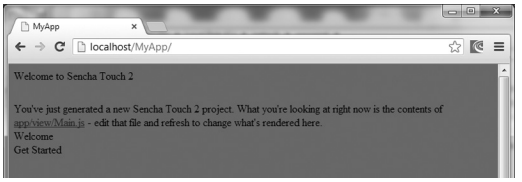


Figure 9.10 The default output that is displayed upon running the `MyApp` application.

Listing 9.1 Default Code in `MyApp/app/view/Main.js` File

```
Ext.define('MyApp.view.Main', {
   extend: 'Ext.tab.Panel',
   xtype: 'main',
   requires: [
      'Ext.TitleBar',
      'Ext.Video'
   ],
   config: {
      tabBarPosition: 'bottom',
      items: [
      {
         title: 'Welcome',
         iconCls: 'home',
         styleHtmlContent: true,
         scrollable: true,
         items: {
            docked: 'top',
            xtype: 'titlebar',
            title: 'Welcome to Sencha Touch 2'
         },
         html: [
            "You've just generated a new Sencha Touch 2 project. What you're
            looking at right now is the ", "contents of <a target = '_blank' href
            = \"app/view/Main.js\">app/view/Main.js</a> - edit that file ", "and
            refresh to change what's rendered here."
         ].join("")
      },
      {
         title: 'Get Started',
         iconCls: 'action',
         items: [
         {
            docked: 'top',
            xtype: 'titlebar',
            title: 'Getting Started'
         },
         {
            xtype: 'video',
            url: 'http://av.vimeo.com/64284/137/87347327.mp4?token =
            1330978144_f9b698fea38cd408d52a2393240c896c',
            posterUrl: 'http://b.vimeocdn.com/ts/261/062/261062119_640.jpg'
         }
         ]
      }
      ]
   }
});
```

Before we begin with understanding the autogenerated Sencha Touch code, let us briefly introduce the Sencha Touch `Component` class that plays a major role in Sencha Touch applications.

Understanding Component Class

All the visual components, like buttons, panels, sliders, toolbars, and so on, that are rendered on the screen are inherited from a base class called `Ext.Component`. This class acts as a building

Table 9.1 Brief Description of the `Ext.Component` Class

Method	*Description*
`addCls` and `removeCls`	Adds or removes a CSS class from a component.
`destroy`	Removes the component from memory.
`disable` and `enable`	Disables or enables the component.
`getHeight`, `getWidth`, and `getSize`	Retrieves the current height, width, or size of the component, respectively. `getSize` returns both height and width.
`setHeight`, `setWidth`, and `setSize`	Changes the height, width, or size of a component.
`show` and `hide`	Shows or hides the component.
`setPosition`	Locates the component at the desired position.
`update`	Update the content of a component.

block for all of the other components used in Sencha Touch applications and also provides configuration settings, methods, properties, and events to manipulate components as the application demands; that is, the components can be displayed, hidden, switched to an enabled or disabled state, resized as per the available screen space, and so on. `Ext.Component` also contains a number of useful methods that are used to get and set properties of any Sencha Touch component. Table 9.1 shows a few of the methods that are supported by the `Ext.Component` class.

Components are placed inside containers. Let us know more about `Container` classes.

Container Classes

Container classes in Sencha Touch, as the name suggests, are the classes where other components, like buttons, input fields, toolbars, and so on, are kept. To render items or components in an application, they need to be placed in a container and arranged through a layout. The container controls height, width, margin, and so on, of the contained components. That is, the components or child objects in a container inherit all the attributes, like height, width, and so forth, of the container object. A few examples of containers include Panel, TabPanel, Toolbar, Carousel, and so on. All the container classes extend the `Ext.Container` class. That is, `Ext.Container` is the base class for all the containers. It provides the basic features that are required in different specific containers. `Ext.Panel` is the default container class used by Sencha Touch that supports docking.

To arrange components in a container, layouts are used. Let us now take a look at them.

Layouts

Layouts are used to resize, arrange, and position the components in a container. The following are the types of layouts that determine how the components in a container have to be arranged:

- **Auto**—Renders one item after another. It is the default layout.
- **Hbox**—Arranges items horizontally in a container, that is, one item beside the other.

- **Vbox**—Arranges items vertically in a container, that is, one item below the other.
- **Card**—Renders items as a card; that is, only one item at the top will be visible and the others will be hidden behind it.
- **Dock**—Handles the docking for panels.
- **Fit**—Renders a single item and automatically expands it to fill the container.

Different configuration options are used in a layout. Table 9.2 shows a brief description of the configuration options used.

Every container in Sencha Touch has a `config` property named `layout` that accepts the type of layout we want to use to arrange and resize the items in a container.

The containers also support the following methods:

- **Query**—Searches for desired items within a container.
- **Update**—Updates the contents of the container.

The following are the two configuration options that are used in a container:

- `flex`—The flex configuration determines the space that a component can use in proportion to the overall layout. For example, if a container has a flex value of 2 and the other has a flex value of 1 in a `vbox` layout, it means that the container with the flex value of 2 will be taller; that is, it will be twice the size of the container with the flex value of 1.

Table 9.2 A Brief Description of the Configuration Options Used in a Layout

Configuration Option	*Description*

`align`	Used to align the specified component in a layout. That is, the components can be aligned to the start, end, or center of the layout. This configuration option can also be used to stretch the specified component to occupy the complete container height or width. For example, the statements given below make the component occupy the complete container height:
	`layout: {`
	`type: 'vbox',`
	`align: 'stretch'`
	`}`
	Valid values for the align configuration options are `center`, `start`, `end`, and `stretch`.

`direction`	Used to lay out items in reverse order. By default, the items are arranged from left to right. On assigning the `reverse` value to the `direction` option, the items will be arranged in reverse order, that is, from right to left. The statements given below lay out the items horizontally, from right to left:
	`layout: {`
	`type: 'hbox',`
	`direction: 'reverse'`
	`}`

- cls—This configuration is used for applying the specified Cascading Style Sheets (CSS) class on the containers. For example, the following configuration option, when used in a container, will apply the styles mentioned in the redStyle class to the container:

```
cls: 'redStyle'
```

Creating Views

The easiest way to create a view in a Sencha Touch application is to use the Ext.create class with an existing component. For example, the following code, when written in the MyApp/app/view/Main.js file (in the MyApp application), will create a simple panel with some HTML inside:

```
Ext.create('Ext.Panel', {
   html: 'Hello World!',
   fullscreen: true
});
```

The code above creates a panel with the HTML content showing the text Hello World! (see Figure 9.11). The configuration option fullscreen resizes the container (panel) to fill the entire screen irrelevant of the device used. Recall that the Ext.Panel is the default container which is usually used to render items on the screen.

A better way to use a container class is to create our custom class that extends the container class followed by instantiating our custom class, as shown in the statements below:

```
Ext.define('MyApp.view.Main', {
   extend: 'Ext.Panel',
   config: {
      html: 'Hello World!',
      fullscreen: true
   }
});
Ext.create('MyApp.view.Main');
```

In the code above, we create a view class named Main that extends the Ext.Panel class container. We use Ext.define for creating a new class. The view class is represented by the MyApp.view.Main convention, where MyApp represents the application name. The config object is defined to specify the config options for the new class. The view class is then instantiated using the Ext.create function. Upon running the application, we get the same text message, Hello World!, on the screen (see Figure 9.11).

To understand how Sencha Touch applications operate, we will begin with the index.html file.

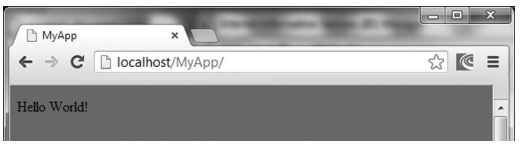


Figure 9.11 the output displayed upon running the MyApp application.

Understanding the `Index.html` File

The `index.html` file includes the JavaScript and CSS files provided in the Sencha Touch package. Even the content of our application is provided in the form of JavaScript. The default `index.html` file that is autogenerated by `Sencha Cmd` in the `MyApp` application is a bit complex to understand in the beginning, so we will create another HTML file called `index2.html` in the `MyApp` folder with simplified content, as shown in Listing 9.2.

Listing 9.2 Code Written in the `index2.html` File

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset = "utf-8">
      <title>Hello World</title>
      <script src = "touch/sencha-touch-debug.js" type = "text/javascript">
      </script> #1
      <link href = "touch/resources/css/sencha-touch.css" rel = "stylesheet" type
      = "text/css"/> #2
      <script src = "my_app.js" type = "text/javascript"></script> #3
   </head>
   <body></body>
</html>
```

Statement 1 includes the debug version of the Sencha Touch JavaScript library in the HTML file. During development and testing, the debug version of the Sencha Touch library is used because it contains comments and documentation that are used in finding errors that occur. After the HTML file is developed and tested, `sencha-touch.js`, that is, the actual Sencha Touch JavaScript library, is included in the HTML file. That is, statement 1 will be replaced by the following line:

```
<script src = "touch/sencha-touch.js" type = "text/javascript"></script>
```

The `sencha-touch.js` library is optimized for production environments and consumes fewer resources, including bandwidth and memory.

Note: Never edit the `sencha-touch-debug.js` or `sencha-touch.js` files.

Statement 2 includes the default CSS file for the Sencha Touch library, `sencha-touch.css` in the HTML file. The Cascading Style Sheet files contain style information for different elements of the page. In statement 3, `my _ app.js` is the JavaScript file that will contain the source code of our application. We will soon create the `my _ app.js` file in the `MyApp` folder and define its code. The main thing to observe in the application above is the opening and closing set of `<body></body>` tags. The output of the code that we will be providing in our JavaScript file, `my _ app.js`, will be displayed through the empty `<body></body>` tag.

Almost all `index.html` files in a Sencha Touch application have the structure shown in Listing 9.2.

Note: To write comments, enclose the text between `<!--` and `-->` symbols. Anything enclosed between `<!--` and `-->` symbols will not be displayed in the browser. The comments are used for increasing the readability of an application.

The Ext object plays a major role in the Sencha Touch application that is created automatically when the Sencha Touch library is loaded. The Ext object provides methods to create an application and its components. Let us discuss the functions that will be used in the JavaScript file(s) of our application.

Using the Ext.application Class

The Ext.application class creates and configures a new Sencha Touch application. This class also enables us to structure our application in the form of the Model View Controller, which is why every Sencha Touch application uses the Ext.application class. While using the Ext.application class, several configuration options are passed to it for configuring the application. The most commonly used configuration options are provided below:

- **Name**—A string that provides the namespace for all the objects contained inside the application. Sencha Touch creates this namespace automatically to define the scope for the application's objects. There should be no space in the name.
- **Launch**—This is a configuration option that specifies a function to be executed when the framework code is loaded and the application is ready to execute.

Note: Sencha Touch uses the namespace Ext for each of its functions to avoid conflict with the user's functions.

We will write the code shown in Listing 9.3 in our JavaScript file, my _ app.js, to display an alert dialog when the application starts.

Listing 9.3 Code Written in the my _ app.js File

```
Ext.require(['Ext.MessageBox']);
Ext.application({
   name: 'WelcomeApp',
   launch: function () {
      Ext.Msg.alert("Title","Hello World!");
   }
});
```

We can see that the alert() function has two parameters: one is the title of the dialog, and the other is the text to be displayed in the alert dialog. To run the index2.html file that we created in the MyApp folder, open the browser and point at the address http://localhost/MyApp/index2.html, followed by the Enter key. An alert dialog will appear displaying the text Hello World!, as shown in Figure 9.12.

We have already learned to use the Ext.Panel container (refer to the "Creating Views" section). Let us now learn how to use the base container class, Ext.Container, to display a HTML text, Hello World!. To do so, modify the JavaScript file my _ app.js to appear as shown in Listing 9.4.

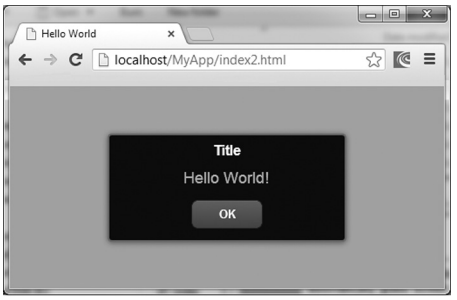


Figure 9.12 The alert dialog displaying the `Hello World!` message.

Listing 9.4 Code Written in the `my _ app.js` File

```
Ext.application({
   name: 'WelcomeApp',
   launch: function () {
      var message = new Ext.Container({
         fullscreen: true,
         html: 'Hello World'
      });
      this.viewport = message;
   }
});
```

In the code above, an application is created called `WelcomeApp`. The application specifies a function that is launched or executed when the application is ready to execute. In the launched function an object of the `Ext.Container` class is created called `message`. The `size` of the container is set to fill up the entire screen. The items are a collection of Sencha Touch components that we want to be included in our container. The items list is enclosed in brackets, and the individual components within the items list are enclosed in curly brackets. An item in the form of HTML text `Hello World!` is rendered in the container. By default, a new application has a viewport that acts as a container to define the views of the application. The application's viewport is set to the `Container` object, and the `message` to display the item, that is, the HTML content, is defined in the container on the screen (see Figure 9.13).

There is one more class called `Ext.setup` that is used to set up an application and execute a function called `onReady` when the document is ready. To use the `Ext.setup` class, modify the JavaScript file `my _ app.js` to appear as shown in Listing 9.5.

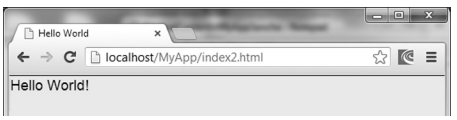


Figure 9.13 The `Hello World!` message is displayed upon running the `index2.html` file.

Listing 9.5 Code Written in the `my _ app.js` File

```
Ext.setup({
   onReady: function() {
      var message = new Ext.Container({
         fullscreen: true,
         items: [{
            html: 'Hello World'
         }]
      });
   }
});
```

Upon running the application, the `Ext.setup` class will set up an application and will execute the `onReady` function, which in turn will display the `Hello World` message on the screen, as shown in Figure 9.13.

Note: The `Ext.application` class provides more features than the `Ext.setup` class while configuring a new application.

Sencha Touch Fields

Sencha Touch supports the standard HTML field types given below:

- `checkboxfield`
- `fieldset`
- `hiddenfield`
- `passwordfield`
- `radiofield`
- `selectfield`
- `textfield`
- `textareafield`

Sencha Touch also supports the following specialized text fields that automatically validate the user's input:

- `emailfield`—Accepts only a valid e-mail address.
- `numberfield`—Accepts only numbers.
- `urlfield`—Accepts only a valid Web URL.

All the fields inherit their height, width, style, and so on, from the container configuration settings. We can optionally apply the following field-specific options to the fields:

- `label`—Used to display a text label with the field.
- `labelAlign`—Used to align the label with the field. That is, the label can be aligned to the `top` or `left` of the field. The default value of this option is `left`.
- `labelWidth`—Used to define the width of the label.
- `name`—Used to assign a name to the field.
- `maxLength`—Used to define the number of characters that can be used in the field.
- `required`—Compels the user to enter some data in the field to submit the form.

One method that is used to create components, containers, and so forth, in a Sencha application is to instantiate their respective classes. Another method is to use `xtype`. Let us learn more.

Using `xtype`

`xtype` acts as shorthand for the class. It does not create the components immediately; instead, it creates them when they are actually required to be displayed on the screen. This technique not only simplifies the task of creating components, but also saves a lot of device memory.

For example, to display a text field, we need to instantiate the `Ext.form.TextField` class as shown below:

```
var nameField = new Ext.form.TextField({
   name: 'username',
   label: 'UserName'
});
```

A text field can also be created through `xtype` as shown below:

```
{
   xtype: 'textfield',
   name: 'username',
   label: 'UserName'
}
```

Event Handling

To implement user interaction, we need to define various event handlers to respond to different events that might occur on different components of the application. The syntax of defining an event handler is as given below:

```
event: event_handler_function
```

Example

The statements below handle the tap event on a button:

```
tap: function(button, e, eOpts) {
   ...........
   ...........
}
```

Different classes support different events. For example, the `Ext.Component` class supports initialize, hide, and show events; the `Ext.Container` class supports add and remove events; the `Ext.Button` class supports tap events; the `Ext.field.Text` class supports focus and blur events; and so on.

Most of the mobile frameworks, including Sencha Touch, support the following types of touch gestures:

- **Tap**—Represents a single touch on the screen.
- **Double-tap**—Represents two quick touches on the screen.
- **Swipe**—Represents moving of a single finger across the screen, from left to right or top to bottom.
- **Pinch or spread**—Represents touching the screen with two fingers and bringing them together or spreading them apart.
- **Rotate**—Represents placing two fingers on the screen and twisting them in a clockwise or counterclockwise direction.

To listen for the occurrence of different events, Sencha Touch uses listeners.

Listeners

The term itself depicts that listeners listen or sense for the occurrence of different events in the components contained in a container and perform responsive actions. Like any configuration option, listeners can be associated with any component in Sencha Touch.

Getting Components

In order to fetch the data entered by the user in different components, we need to get or access the components used in an application. To get components that exist in an application, Sencha Touch provides the following two functions:

- `Ext.getCmp()`—Gets the component with the supplied ID. For example, the statement given below gets the component with the ID `username`:

```
var username = Ext.getCmp('username');
```

- `Ext.ComponentQuery()`—Gets single or multiple components that are related to the supplied component. For example, the statement given below gets all toolbars in an application:

```
var toolbars = Ext.ComponentQuery.query('toolbar');
```

Let us apply the knowledge gained thus far in creating Sencha Touch applications.

Creating a Welcome Application

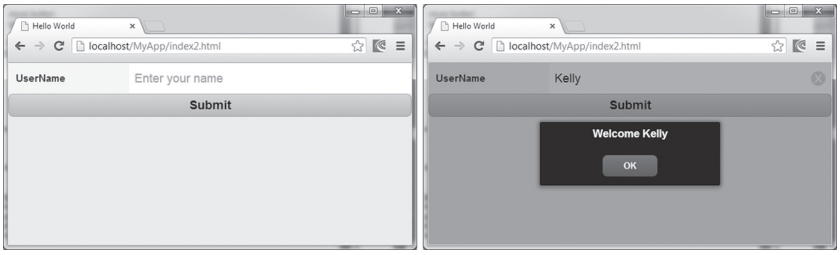
Let us create a welcome application that prompts the user to enter a name. After entering a name, when the user clicks the button, a welcome message is displayed along with the user's name. The application will require a text field and a button, and to display them in our application, we will make use of `xtype`. Modify the `my _ app.js` in the `MyApp` folder to appear as shown in Listing 9.6.

Listing 9.6 Code Written in the `my _ app.js` File

```
Ext.require(['Ext.MessageBox']);
Ext.setup({
   onReady: function() {
      var message = new Ext.Container({
         fullscreen: true,
         items: [
         {
            xtype: 'textfield',
            name : 'username',
            label: 'UserName',
            id: 'username',
            placeHolder: 'Enter your name'
         },
         {
            xtype:'button',
            text:'Submit',
            listeners: {
               tap: function(button, e, eOpts) {
                  var name = Ext.ComponentQuery.query('#username')[0].getValue();
                  Ext.Msg.alert("Welcome "+ name);
               }
            }
         }
         ]
      });
   }
});
```

In the code above, we have used a placeholder in the text field. Placeholders are supported in most of the form fields and are used to display initial text in the fields to guide the user on what has to be entered in the respective fields. The initial text displayed through placeholders in the fields disappears when the fields are focused and some data are entered in them. We can see that a listener is associated with the button that listens or waits for the occurrence of a tap event on the button.

Upon running the `index2.html` file in the `MyApp` folder, we get a text field and a button control on start-up, as shown in Figure 9.14a. When the button is tapped, an event handler function is executed that retrieves the username entered in the text field of the ID `username` and displays it in the alert dialog with a welcome message (see Figure 9.14b).



(a) (b)

Figure 9.14 (a) A text field and button are displayed on application start-up. (b) A welcome message is displayed upon clicking the button.

The welcome application above was created by using two files, `index2.html` and `my_app.js`. The application did not follow the MVC architecture format. What if we want to create the same welcome application in the MVC architecture format? Let us see how it can be done.

Welcome Application in MVC Format

In MVC architecture, different JavaScript files have to be created that represent the `model`, `view`, and `controller` of the application, and these files have to be kept in the respective subfolders of the `app` folder of the application.

Let us name our welcome application `WelcomeApp`. So, in the Web folder (`C:\inetpub\wwwroot`), create a directory called `WelcomeApp`. In the `WelcomeApp` directory, copy the `touch` folder (refer to Figure 9.9) from the `MyApp` directory, as it contains all the Sencha Touch package files that will be required in the application. Also, create a folder named `app` in the `WelcomeApp` directory. Inside the `app` folder, create five subfolders named `controller`, `model`, `profile`, `store`, and `view`. Figure 9.15 shows the pattern of files and folders that we will be creating in this application.

Create a file `index.html` with the code as shown in Listing 9.7 in the root of the application, that is, in the `WelcomeApp` folder.

Listing 9.7 Code Written in the `index.html` File

```
<!DOCTYPE html>
<html>
<head>
   <title>Welcome App</title>
   <script src = "touch/sencha-touch-debug.js" type = "text/javascript"></script>
   <link href = "touch/resources/css/sencha-touch.css" rel = "stylesheet" type =
   "text/css"/>
   <script src = "app.js" type = "text/javascript"></script>
</head>
<body></body>
</html>
```

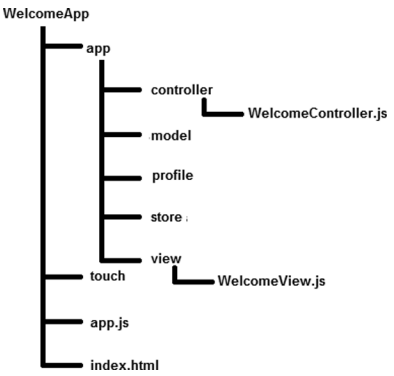


Figure 9.15 The files and folders of the `WelcomeApp` application in the MVC architecture format.

As expected, the `index.html` file includes the debug version of the Sencha Touch JavaScript library, the default CSS file for the Sencha Touch library, and our custom JavaScript file, `app.js`, that will contain our application code. The file includes a closing set of `<body></body>` tags that will be used to display the output of the application.

Create our custom JavaScript file called `app.js` in the root folder of the application and write the code as shown in Listing 9.8.

Listing 9.8 Code Written in the `app.js` File

```
Ext.application({
   name: "WelcomeApp",
   controllers: ["WelcomeController"],
   views: ['WelcomeView'],
   launch: function () {
      var welcomePanel = Ext.create('Ext.Panel', {
         layout: 'fit',
         items: [
         {
            xtype: 'welcomeform'
         }
         ]
      });
      Ext.Viewport.add(welcomePanel);
   }
});
```

In the code above, we can see that our application is named `WelcomeApp`. The application defines the controller file as `WelcomeController` and the view filename as `WelcomeView`. The application also specifies a function that is launched or executed when the application is ready to execute. In the launched function a container is created; that is, an object of the `Ext.Panel` class is created called `welcomePanel`. The size of the Panel container is set to automatically expand to fill up the entire screen. The item that is displayed in the Panel container is the `welcomeform` item. The details of the `welcomeform` item will be provided in the controller of the application. Finally, the application's viewport is set to the container object, `welcomePanel`, to display the item, that is, UI is defined in `welcomeform`.

Styling Buttons

In this application, we will use a button that, when clicked, will perform the desired action. Buttons in Sencha Touch have a `ui` configuration setting that can be used to style buttons. The options used in the `ui` configuration setting are provided below:

- **Normal**—Displays the default button.
- **Back**—Displays a back button, that is, a button pointing to the left side.
- **Round**—Displays a rounded button.
- **Small**—Displays a smaller button than the default.
- **Action**—Displays a button that is brighter than the default button.
- **Forward**—Displays a button pointing to the right side.

In addition to text, we can also display icons in buttons. Let us take a look at this now.

Displaying Icons in Buttons

To make it easier to remember, we can always display an icon along with the text in a button. The properties that can be used for defining an icon in a button are given below:

- `icon`—Used to define an image to be used as an icon.
- `iconCls`—Used to define an icon through a CSS class.
- `iconAlign`—Used to align the icon with respect to the button text. The valid values are `top`, `bottom`, `right`, and `left`, where `left` is the default value.

The following statements display a normal button with the text `Home` and with the icon defined in the `home` CSS class:

```
{
   xtype: 'button',
   ui : 'normal',
   text: 'Home',
   iconCls: 'home',
   iconAlign: 'right'
}
```

The icon will be aligned to the right of the button text. To define a text field and a button in the application's view, we will write the code as shown in Listing 9.9 in the `app/view/WelcomeView.js` file. Recall, `app.js` file (refer to Listing 9.8) describes the view of the application as `WelcomeView.js`.

Listing 9.9 Code Written in the `WelcomeView.js` File

```
Ext.define("WelcomeApp.view.WelcomeView", {
   extend: "Ext.form.FormPanel",
   alias: "widget.welcomeform",
   itemId: 'welcomeForm',
   config: {
      title: 'Welcome Form',
      items: [
      {
         xtype: 'fieldset',
         itemId: 'NameFieldset',
         items: [
         {
            xtype: 'textfield',
            label: 'User Name',
            name: 'username',
            id: 'username',
            required: true,
            placeHolder: 'Enter your name'
         }
         ]
      },
      {
         xtype: 'button',
         id: 'submitButton',
         ui: 'action',
         text: 'Submit',
```

```
            action: 'submit'
        }
        ]
    }
});
```

We can see that the `Ext.form.FormPanel` class is extended to create a form panel with the ID `welcomeForm`. The title assigned to the form is `Welcome Form`. In the form, a text field is defined with the ID `username`. The text field will be marked with an asterisk to indicate that it is a required field and cannot be left blank. A placeholder text, `Enter your name`, will be displayed in the text field to inform the user that a name has to be entered in the text field. A button with the text `Submit` is also defined. The ID assigned to the button is `submitButton`, and the `action` value is set to `submit`; that is, when the button is clicked, the `action` and its assigned value are passed to the controller to perform the desired task. Also, in the code above, we have used the `fieldset xtype`. Recall, from Chapter 6, the `fieldset` element is used to logically group together elements in a form.

Next, we need to define the controller of the application to do the following tasks:

- Implement interaction with the form
- Listen for the click or tap event on the button
- Define the function to be executed to perform the task when the button is clicked

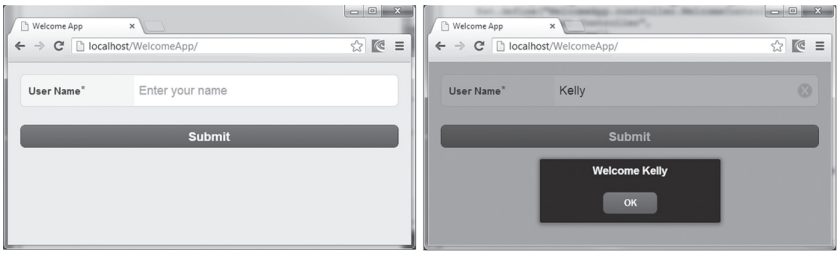
To do the tasks above, the code as shown in Listing 9.10 is written in the `app/controller/WelcomeController.js` file.

Listing 9.10 Code Written in the `WelcomeController.js` File

```
Ext.require(['Ext.MessageBox']);
Ext.define("WelcomeApp.controller.WelcomeController", {
    extend: "Ext.app.Controller",
    views: ['WelcomeView'],
    config: {
        refs: {
            welcomeForm: "#welcomeForm"
        },
        control: {
            'button[action = submit]': {
                tap: "displayMessage"
            }
        }
    },
    displayMessage: function (button) {
        var name = Ext.ComponentQuery.query('#username')[0].getValue();
        Ext.Msg.alert("Welcome "+ name);
    }
});
```

Upon running the application, a text field and a button will be displayed as shown in Figure 9.16a. After entering a name in the text field, when the user clicks the `Submit` button, a welcome message will be displayed along with the user's name, as shown in Figure 9.16b.

After creating a welcome application, let us create a login form application.



(a) (b)

Figure 9.16 (a) The text field and a button are displayed upon application start-up. (b) A welcome message along with the user's name are displayed.

Creating a Login Form

In the login form application, the user will be prompted to enter the username and password. If the username and password entered are `guest` and `gold`, respectively, then a welcome message will be displayed; otherwise, the `Invalid username/password` message will be displayed.

So, let us create a new directory called `LoginApp` in the Web folder (`C:\inetpub\wwwroot`). In the `LoginApp` directory, copy the `touch` folder that contains the Sencha Touch package files. Also, create a folder named `app` in the `LoginApp` directory. Inside the `app` folder create five subfolders by the names `controller`, `model`, `profile`, `store`, and `view`. Create a file `index.html` with the code as shown in Listing 9.11 in the root of the application, that is, in the `LoginApp` directory.

Listing 9.11 Code Written in the `index.html` File

```
<!DOCTYPE html>
<html>
<head>
   <title>Login Form</title>
   <script src = "touch/sencha-touch-debug.js" type = "text/javascript"></script>
   <link href = "touch/resources/css/sencha-touch.css" rel = "stylesheet" type =
   "text/css"/>
   <script src = "app.js" type = "text/javascript"></script>
</head>
<body></body>
</html>
```

The `index.html` file includes the debug version of the Sencha Touch JavaScript library, the default CSS file for the Sencha Touch library, and our custom JavaScript file, `app.js`, that will contain our application code. The file includes a closing set of `<body></body>` tags that will be used for displaying the output of the application. Create our custom JavaScript file called `app.js` in the root folder of the application and write the code as shown in Listing 9.12.

Listing 9.12 Code Written in the `app.js` File

```
Ext.application({
   name: "LoginApp",
   controllers: ["LoginController"],
```

```
    views: ['LoginView'],
    launch: function () {
        var loginPanel = Ext.create('Ext.Panel', {
            layout: 'fit',
            items: [
            {
                xtype: 'mylogin'
            }
            ]
        });
        Ext.Viewport.add(loginPanel);
    }
});
```

In the code above, we can see that the application is named `LoginApp`, the controller is defined by the name `LoginController`, and the view is defined by the name `LoginView`. In the function that is launched when the application is ready, a Panel container is created called `LoginPanel`. The size of the Panel container is set to automatically expand to fill up the entire screen. The item that is displayed in the Panel container is `mylogin`. The details of the `mylogin` item will be provided in the controller of the application. Finally, the application's viewport is set to the container object, `loginPanel`, to display the item, that is, login form.

To define a text field, a password field, and a button in the application's view, we will write the code as shown in Listing 9.13 in the `app/view/LoginView.js` file.

Listing 9.13 Code Written in the `LoginView.js` File

```
Ext.define("LoginApp.view.LoginView", {
    extend: "Ext.form.FormPanel",
    alias: "widget.mylogin",
    id: 'loginFormPanel',
    config: {
        margin: '0 auto',
        name: 'loginform',
        frame: true,
        url: 'authenticate.php',
        title: 'Login',
        items: [
        {
            xtype: 'fieldset',
            itemId: 'LoginFieldset',
            margin: '10 auto 0 auto ',
            title: '',
            items: [
            {
                xtype: 'textfield',
                label: 'User Name',
                name: 'username',
                required: true,
                placeHolder: 'Username'
            },
            {
                xtype: 'passwordfield',
                label: 'Password',
                name: 'password',
                required: true,
                placeHolder: 'Password'
            }
```

```
            ]
        },
        {
            xtype: 'button',
            id: 'loginButton',
            margin: '25 auto 0 auto ',
            style: '',
            maxWidth: 200,
            ui: 'action',
            width: '',
            iconCls: 'user',
            iconMask: true,
            text: 'Login',
            action: 'login'
        }
        ]
    }
});
```

We can see that the `Ext.form.FormPanel` class is extended to create a form panel with the ID `loginFormPanel`. The title assigned to the form is `loginFormPanel`. In the form, a text field is defined with the name `username`, a password field is defined with the name `password`, and a button is defined with the ID `loginButton`. The text displayed on the button is `Login`, and its action value is set to `login`. When the button is clicked, the action and its assigned value are passed to the controller to perform the desired task.

It is now time to define the controller of the application to do the following tasks:

- Implement interaction with the login form.
- Listen for the click or tap event on the button.
- Direct the data entered in the login form to the Hypertext Preprocessor (PHP) file for authentication.
- Observe the result returned by the PHP file. If the result returned by the PHP file is a `success`, that is, if the username and password match the desired texts, display a welcome message on the screen.
- Display the message `Invalid username/password` if the result returned by the PHP file is a `failure`; that is, the username or password entered by the user does not match the specified text.

To do the tasks above, the code as shown in Listing 9.14 is written in the `app/controller/LoginController.js` file.

Listing 9.14 Code Written in the `LoginController.js` File

```
Ext.require(['Ext.MessageBox']);
Ext.define("LoginApp.controller.LoginController", {
    extend: "Ext.app.Controller",
    views: ['LoginView'],
    config: {
        refs: {
            loginForm: "#loginFormPanel"
        },
        control: {
            'button[action = login]': {
                tap: "authenticateUser"
```

```
            }
        }
    },
    authenticateUser: function (button) {
        this.getLoginForm().submit({
            url: 'authenticate.php',
            method: 'POST',
            success: function (form, result) {
                Ext.Msg.alert("Success", "Welcome ");
            },
            failure: function (form, result) {
                Ext.Msg.alert("Error", "Invalid username/password");
            }
        });
    }
});
```

We need to create a PHP file to perform the following tasks:

- Access the username and password entered in the login form
- Compare the entered name and password with the desired texts
- Return the comparison results

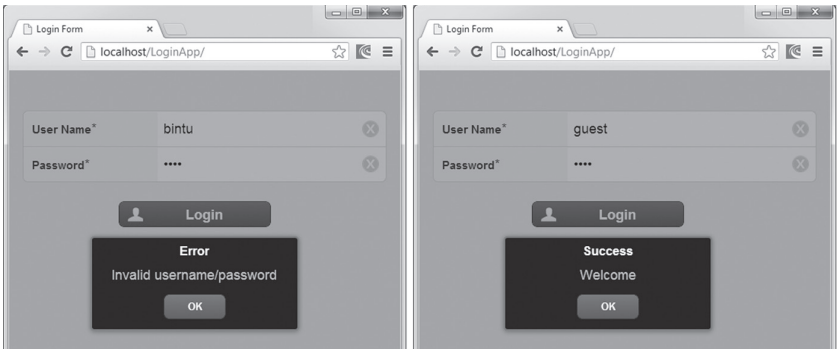
To do the tasks above, write the code shown in Listing 9.15 in the `authenticate.php` file.

Listing 9.15 Code Written in the `authenticate.php` File

```
<?php
    $usr = $_REQUEST['username'];
    $pw = $_REQUEST['password'];
    header('Content-Type: application/json');
    if($usr = = 'guest' && $pw = = 'gold'){
        echo '{"success":true, "msg":'.json_encode('This User is authorized').'}';
    }else{
        echo '{"success":false, "msg":'.json_encode('This User is NOT
        authorized').', "errors" : {"password" :'.json_encode('Password is
        required').'}'.', "pwd" :'.json_encode($pw).'}';
    }
?>
```

In the code above, we can see that the username and password entered in the login form are accessed and assigned to variables `usr` and `pw`, respectively. The data in the `usr` and `pw` variables are matched with the text `guest` and `gold`, respectively. If the user enters the username and password as `guest` and `gold`, then a `success` message is echoed; otherwise, a `failure` message is echoed. The result, that is, the `success` or `failure` message, is used in the controller file to inform the user whether or not the login was successful.

Upon running the application, a text field, a password field, and a button will be displayed. Upon entering the incorrect username or password, the message `Invalid username/password` will be displayed, as shown in Figure 9.17a. After entering the username as `guest` and the password as `gold`, when the button is clicked, a welcome message will be displayed, as shown in Figure 9.17b.



(a) (b)

Figure 9.17 (a) An invalid username/password message is displayed if an incorrect username or password are entered. (b) A welcome message is displayed if the correct username and password are entered.

Creating a List Application

In this application, we will learn how to display a few products through a list, and when any item from the list is chosen, the name of the selected item will be displayed through an alert dialog. The idea behind developing this application is to show you how certain options can be displayed via a list and how event handling is performed upon selecting an item from the list.

So, now we create a new directory called `ListApp` in the Web folder (`C:\inetpub\wwwroot`). Again, in the `ListApp` directory, copy the `touch` folder that contains the Sencha Touch package files. Also, create a folder named `app` in the `ListApp` directory. Inside the `app` folder create five subfolders by the name `controller`, `model`, `profile`, `store`, and `view`. Create a file called `index.html` with the code as shown in Listing 9.16 in the root folder of the application, that is, in the `ListApp` directory.

Listing 9.16 Code Written in the `index.html` File

```
<!DOCTYPE html>
<html>
<head>
   <title>List Application</title>
   <script src = "touch/sencha-touch-debug.js" type = "text/javascript"></script>
   <link href = "touch/resources/css/sencha-touch.css" rel = "stylesheet" type =
   "text/css"/>
   <script src = "app.js" type = "text/javascript"></script>
</head>
<body></body>
</html>
```

The `index.html` file includes the debug version of the Sencha Touch JavaScript library, the default CSS file for the Sencha Touch library, and our custom JavaScript file, `app.js`, which will contain our application code. The file includes a closing set of `<body></body>` tags to display the output of the application.

Create our custom JavaScript file called `app.js` in the root folder of the application and write the code as shown in Listing 9.17.

Listing 9.17 Code Written in the `app.js` File

```
Ext.application({
   name: 'ListApp',
   views : ['ProductsList'],
   controllers: ['ListController'],
   launch: function() {
      Ext.create('Ext.Container', {
         fullscreen: true,
         layout: 'fit',
         items: [{
            xtype: 'productsList'
         }]
      });
   }
});
```

In the code above, we can see that the application is named `ListApp`, the controller is defined by the name `ListController`, and the view is defined by the name `ProductsList`. In the function that is launched when the application is ready, a container is created. The size of the container is set to automatically expand to fill up the entire screen. The item that is displayed in the container is the `productsList` item. The details of the `productsList` item will be provided in the controller of the application.

To define a list we need to extend the `Ext.dataview.List` class. To define a list and the items to be displayed through it, the code shown in Listing 9.18 is written in the `app/view/ProductsList.js` file.

Listing 9.18 Code Written in the `ProductsList.js` File

```
Ext.define('ListApp.view.ProductsList', {
   extend: 'Ext.dataview.List',
   alias : 'widget.productsList',
   config: {
      data: [
         {text: 'Laptop'},
         {text: 'Camera'},
         {text: 'CellPhone'},
         {text: 'Tablet'}
      ],
      itemTpl: '{text}'
   }
});
```

We can see that the data are defined to keep the names of the products that we want to be displayed through the list. The list is assigned an alias, `productsList`. We want this when any product displayed through the list is selected, its name should be displayed through the alert dialog. To do so, write the code as shown in Listing 9.19 in the `app/controller/ListController.js` file.

Listing 9.19 Code Written in the `ListController.js` File

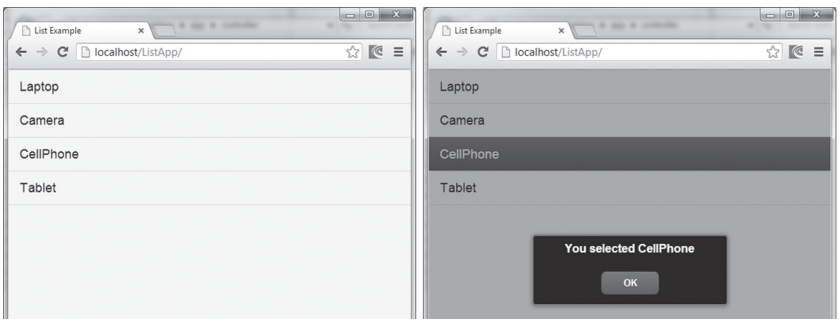
```
Ext.require(['Ext.MessageBox']);
Ext.define('ListApp.controller.ListController', {
   extend : 'Ext.app.Controller',
   config: {
      profile: Ext.os.deviceType.toLowerCase(),
      control: {
         'productsList': {
            itemtap: 'onSelectRow'
         }
      }
   },
   onSelectRow: function(view, index, target, record, event) {
      Ext.Msg.alert("You selected "+record.get('text'));
   }
});
```

We can see that when any item that is displayed through the list is clicked or tapped, the `onSelectRow` function will be invoked, which in turn displays the name of the selected product through the alert dialog. Upon running the application, a few product names will be displayed through a list, as shown in Figure 9.18a. Upon selecting any product from the list, its name will be displayed via the alert dialog, as shown in Figure 9.18b.

Navigating Using Tabs

In this application, we will learn to create two panels called `Home` and `Clients` and will navigate between them using a tab panel. That is, we will create two tabs in the tab panel called `Home` and `Clients`, and when the `Home` tab is clicked, we will be navigated to the `Home` panel. Similarly, when the `Clients` tab is selected from the tab panel, the `Clients` panel will open up.

So, let us create a new directory called `NavigationApp` in the Web folder (`C:\inetpub\wwwroot`). Again, in the `NavigationApp` directory, copy the `touch` folder that contains the Sencha Touch package files. Also, create a folder named `app` in the `NavigationApp` directory. Inside the `app` folder, create five subfolders by the names `controller`, `model`, `profile`,



(a) (b)

Figure 9.18 (a) A few products are displayed through a list. (b) A selected product is displayed through an alert dialog.

store, and view. Create a file called index.html with the code as shown in Listing 9.20 in the root of the application, that is, in the NavigationApp directory.

Listing 9.20 Code Written in the index.html File

```
<!DOCTYPE html>
<html>
<head>
   <title>Navigation App</title>
   <script src = "touch/sencha-touch-debug.js" type = "text/javascript"></script>
   <link href = "touch/resources/css/sencha-touch.css" rel = "stylesheet" type =
   "text/css"/>
   <script src = "app.js" type = "text/javascript"></script>
</head>
<body></body>
</html>
```

As expected, the index.html file includes the debug version and default CSS file of the Sencha Touch JavaScript library and our custom JavaScript file, app.js, which will contain our application code. A closing set of <body></body> tags is defined to display output of the application. Create our custom JavaScript file called app.js in the root folder of the application and write the code as shown in Listing 9.21.

Listing 9.21 Code Written in the app.js File

```
Ext.application({
   name: 'NavigationApp',
   requires: ['NavigationApp.view.Viewport'],
   views: ['Home','Clients'],
   launch : function() {
      Ext.create('NavigationApp.view.Viewport');
   }
});
```

In the code above, we can see that the application is named NavigationApp, and the two views are defined by the names Home and Clients. In the function that is launched when the application is ready, an instance of the view, Viewport, is created. To define a Home panel, the code shown in Listing 9.22 is written in the app/view/Home.js file.

Listing 9.22 Code Written in the Home.js File

```
Ext.define('NavigationApp.view.Home', {
   extend: 'Ext.Panel',
   xtype: 'homepanel',
   config: {
      title: 'Home',
      iconCls: 'home',
      items: [
      {
         xtype: 'toolbar',
         title: 'Home'
      },
      {
```

```
            html: 'Home Panel'
        }
        ]
    }
});
```

We can see that the `Ext.Panel` class is extended to define the `Home` panel. A toolbar is displayed in the panel, the title `Home` is displayed in the panel, and an HTML text, `Home Panel`, is also set to be displayed in the panel. To define another panel, `Clients`, the code shown in Listing 9.23 is written in the `app/view/Clients.js` file.

Listing 9.23 Code Written in the `Clients.js` File

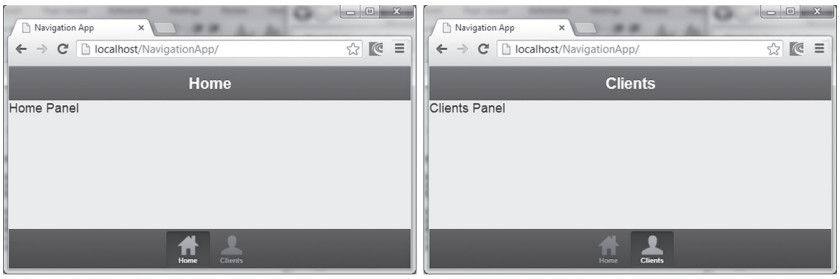
```
Ext.define('NavigationApp.view.Clients', {
    extend: 'Ext.Panel',
    xtype: 'clientpanel',
    config: {
        title: 'Clients',
        iconCls: 'user',
        items: [
        {
            xtype: 'toolbar',
            title: 'Clients'
        },
        {
            html: 'Clients Panel'
        }
        ]
    }
});
```

Like the `Home` panel, the `Clients` panel also displays a toolbar, a title, `Clients`, and an HTML text, `Clients Panel`.

Next, we need to define a view, `Viewport.js`. In the `Viewport` view, we will define a tab panel and position it at the bottom of the screen. In the tab panel, we will display two tabs, `Home` and `Clients`, which, when clicked, will open the respective panel. To do so, write the code as shown in Listing 9.24 in the `app/view/Viewport.js` file.

Listing 9.24 Code Written in the `Viewport.js` File

```
Ext.define('NavigationApp.view.Viewport',{
    extend: 'Ext.tab.Panel',
    config: {
        tabBarPosition: 'bottom',
        fullscreen: true,
        layout: 'card',
        items: [
        {
            xtype: 'homepanel'
        },
        {
            xtype: 'clientpanel'
        }
        ]
    }
});
```



(a) (b)

Figure 9.19 (a) The `Home` panel appears upon application start-up. (b) The `Clients` panel appears upon selecting the `Clients` tab.

Upon running the application, the `Home` panel will open by default, showing the text `Home Panel` on the screen. At the bottom of the screen, a tab panel will appear showing two tabs, `Home` and `Clients`, as shown in Figure 9.19a. Upon selecting the `Clients` tab from the tab panel, the `Clients` panel will open showing the text `Clients Panel` on the screen, as shown in Figure 9.19b.

Summary

In this chapter, we learned to use the Sencha Touch framework for developing mobile applications that are compatible with Android, iOS, and BlackBerry devices. We learned the Model View Controller (MVC) architecture format that is followed by the Sencha Touch applications. We also learned to enable Web servers, install Sencha Touch 2 SDK, and use the Sencha Cmd tool. We detailed the folder organization of the Sencha Touch applications. We also learned how to use the component class, container classes, layouts, `xtypes`, and event handling. We investigated the step-by-step procedure for creating different applications, including the welcome application, login form, list application, and navigating using tabs.

In this book, I have tried to keep things easy to understand. I hope you agree. You now have all the necessary information to solve different issues that you might come across while building and maintaining cross-platform mobile applications using PhoneGap Build.

Have fun creating your own applications, and thanks for reading!

Appendix A: Setting Up an Android Environment

In this application, we will learn to set up an Android environment for developing PhoneGap applications.

Setting Up an Android Environment

For setting up an Android environment for developing applications, we need to install the following software:

- Java SE Development Kit (JDK)—Can be downloaded from `http://oracle.com/technetwork/java/javase/downloads/index.html`.
- Eclipse Integrated Development Environment (IDE)—Can be downloaded from `http://www.eclipse.org/downloads/`.
- Android Platform Software Development Kit (SDK) Starter Package—Can be downloaded from `http://developer.android.com/sdk/index.html`.
- Android Developer Tools (ADT) plug-in—Can be downloaded from `http://developer.android.com/sdk/eclipse-adt.html`. The plug-in contains project templates and Eclipse tools that help in creating and managing Android projects.

Note: The Android application can also be developed through Android Studio—a new development environment. Android Studio can be downloaded from the following URL: http://developer.android.com/sdk/installing/studio.html. But because at the time of this writing, Android Studio is available as only an "early access preview" and few of its features are either incomplete or not yet implemented, I stick to using the Eclipse with ADT Plugin.

The Android SDK includes the core SDK tools, which are used to download the rest of the SDK components. This means that after installing the Android SDK Tools, we need to install Android platform tools and the other components that are required for developing Android applications. Go to `http://developer.android.com/sdk/index.html` and download the

package by selecting the link for your operating system. For Windows users, the provided .exe file is named `installer _ r21-windows.exe`. After downloading the file, double-click it to initiate the installation process. The Android SDK Manager window will open. The dialog boxes that you will see now are from the Windows installer, and the screens may vary from other operating system installers.

The first screen is a welcome screen. Select the `Next` button to move to the next screen. Because Android SDK requires the Java SE Development Kit for its operation, it checks for the presence of JDK on your computer. If JDK is already installed, you will see the screen shown in Figure A.3 later. If JDK is not found, it will display a button with the caption `Visit java.oracle.com`, which you can use to download and install JDK. Upon selecting the button, you will be navigated to `http://www.oracle.com/technetwork/java/javase/downloads/index.html`, which shows links to download Java Platform, Standard Edition, Java SE Development Kit (JDK) bundles, and additional resources. The latest version of Java available at the time of this writing is JDK version 1.7. Select the JDK link that suits your platform (Windows, Linux, or Mac) and double-click the downloaded file to begin JDK installation. You will probably see a `Security Warning` dialog box asking whether you want to run or cancel the execution of the file. Select the `Run` button to initiate JDK installation. The first screen that you see is a Java setup wizard welcome screen. Select the `Next` button to see the `Custom Setup` dialog box for selecting optional JDK features that you want to install, as shown in Figure A.1.

Three categories of features (Development Tools, Source Code, and Public JRE) are displayed, and you can select from the respective drop-down lists to choose the list of features in each category you wish to install. The dialog box also asks for a drive location where you want to install the JDK. The default location displayed is `C:\Program Files\Java\jdk1.7.0 _ 09\`, but you can use the `Change` button to select another location. Let us keep the default settings and click the `Next` button to continue. The selected program features will be installed, followed by a dialog box that prompts for the destination folder to install the runtime environment (JRE), as shown in Figure A.2.

The dialog box displays the default location for installing JRE (`C:\Program Files\Java\jre7`). Use the `Change` button to place the program elsewhere. Let us keep the default location and select the `Next` button to continue. The Java files will be copied and installed on your

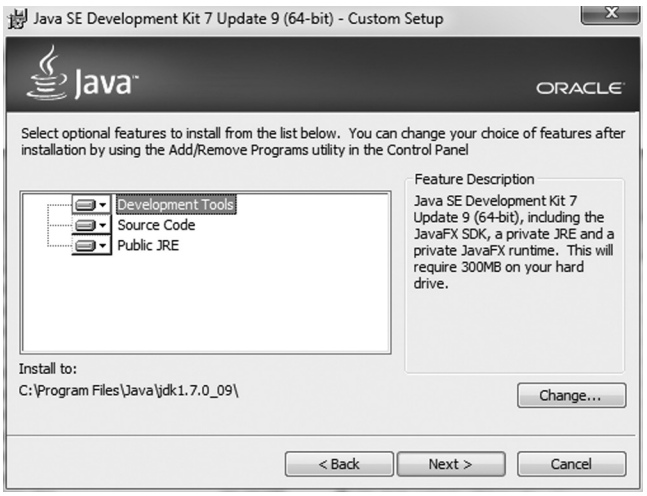


Figure A.1 The `Java Setup` dialog box.

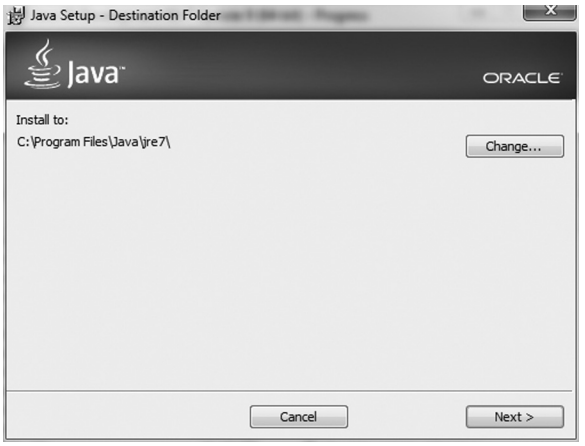


Figure A.2 The dialog box prompting for the JRE installation location.

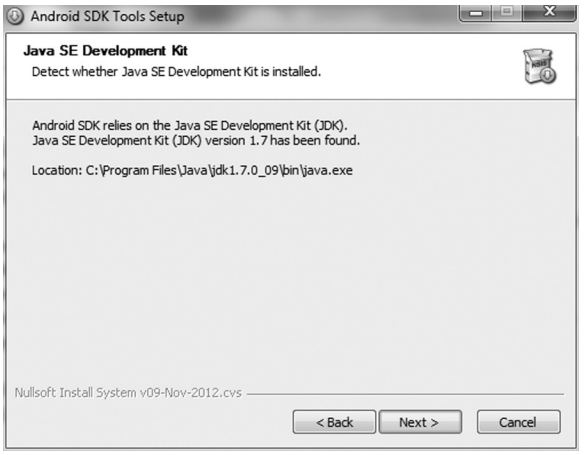


Figure A.3 The dialog box informing you that JDK is already installed on the computer.

machine. If the installation is successful, a confirming dialog box is displayed. Select the `Finish` button to exit the wizard. After Java installation, the `Android SDK Tools` setup wizard will automatically resume.

If Java is already installed on your computer before beginning with the Android SDK installation, the wizard will detect its presence and display the version number of the JDK found on the machine, as shown in Figure A.3.

Select the `Next` button. You will get a dialog box asking you to choose the users for which the Android SDK is being installed. The following two options will be displayed in the dialog box:

- Install for anyone using the computer
- Install just for me

Let us select the option `Install for anyone using this computer`, followed by clicking `Next`. The next dialog prompts us for the location to install the Android SDK Tools,

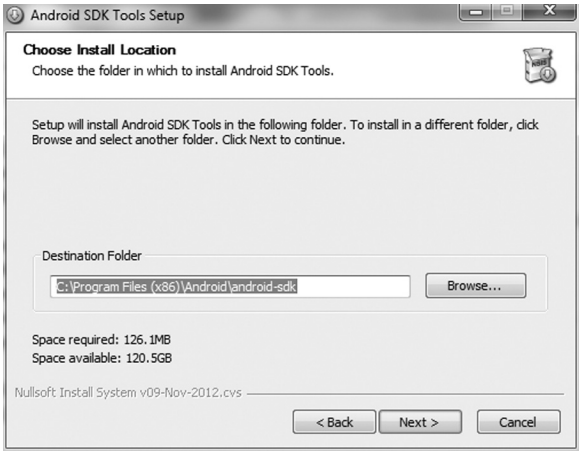


Figure A.4 The dialog box to specify the Android SDK Tools installation location.

as shown in Figure A.4. The dialog also displays the default directory location for installing the Android SDK Tools, `C:\Program Files (x86)\Android\android-sdk`, which you can change by selecting the `Browse` button. Keep the default directory for installing the Android SDK Tools unchanged, and then select the `Next` button to continue.

The next dialog box asks you to specify the `Start Menu` folder where you want the program's shortcuts to appear. A default folder appears called `Android SDK Tools`. If you do not want to make a `Start Menu` folder, select the `Do not create shortcuts` checkbox. Let us create the `Start Menu` folder by keeping the default folder name and selecting the `Install` button to begin the installation of the Android SDK Tools. After all the files have been downloaded and installed on the computer, select the `Next` button. The next dialog box tells you that the `Android SDK Tools` setup wizard is complete and the Android SDK Tools have successfully installed on the computer. Select the `Finish` button to exit the wizard, as shown in Figure A.5.

Note that the checkbox `Start SDK Manager (to download system images)` is checked by default. It means that after the `Finish` button is clicked, the Android SDK Manager,

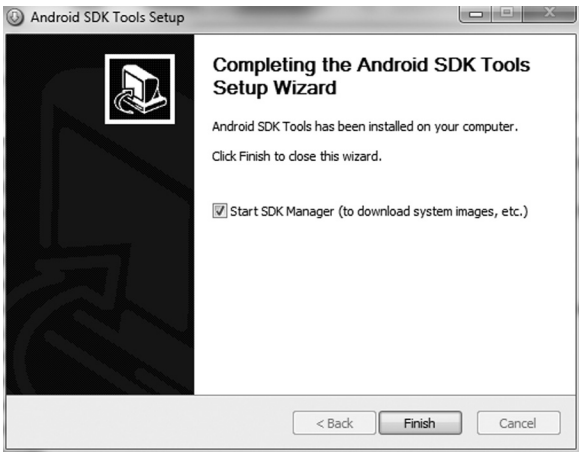


Figure A.5 The successful installation of the Android SDK Tools dialog box.

one of the tools in the Android SDK Tools package, will be launched. The Android SDK is installed in two phases; the first phase is the installation of the SDK, which installs the Android SDK Tools, and the second phase is the installation of the Android platforms and other components.

Adding Platforms and Other Components

In this step you will see how to use the Android SDK Manager to download and install the important SDK packages required for the development environment. The Android SDK Manager (Figure A.6) that opens up shows the list of all the packages and their installation status. The dialog box shows that the Android SDK Tools package is already installed on the machine. To install any other package, you just need to check its checkbox. The Android SDK Manager recommends a platform by checking the `Android 4.2 (API 17)` and `Google USB Driver` packages by default.

You can check more packages and uncheck existing packages to determine which application programming interfaces (APIs) you wish to install. Because you wish to work with the latest Android API, leave the default selected and choose the `Install 7 packages` button at the bottom to initiate installation. The next dialog box you see shows the list of the packages that you have selected to install, their descriptions, and license terms. You need to select the `Accept All` option, followed by the `Install` button to begin installation, as shown in Figure A.7.

An `Android SDK Manager Log` window appears showing the downloading and installation progress. It also shows the list of packages that have been loaded, the Android SDK platform tools that have been downloaded and installed on the machine, and the ones that are still being downloaded (see Figure A.8a). After selecting the `Close` button, the next dialog window is the `ADB Restart` window that provides information about updates and asks whether you wish to

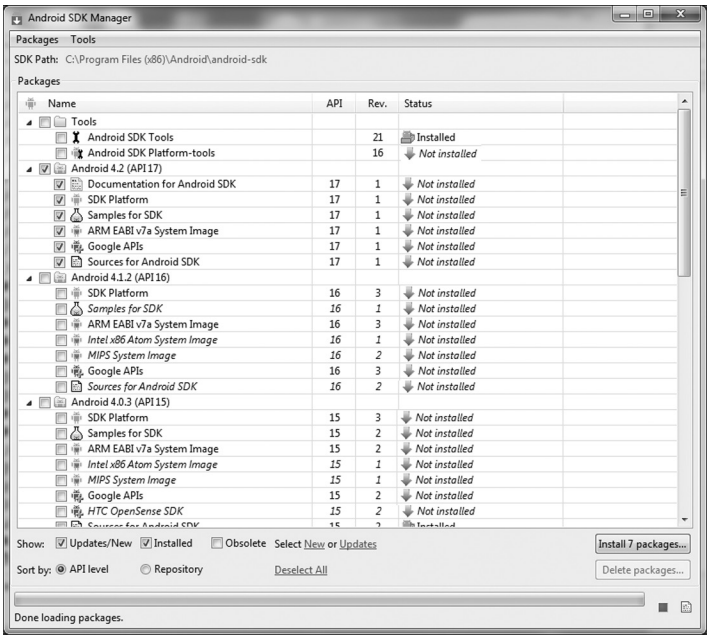


Figure A.6 The Android SDK Manager showing the list of packages and their current status.

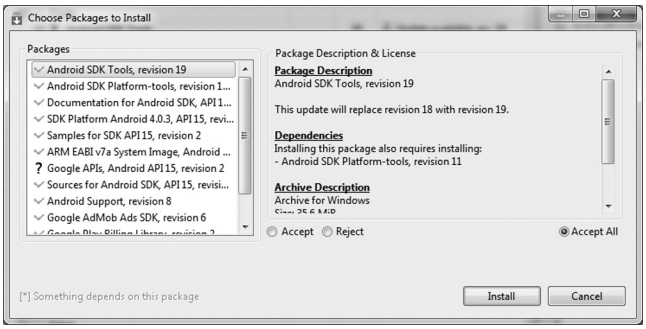
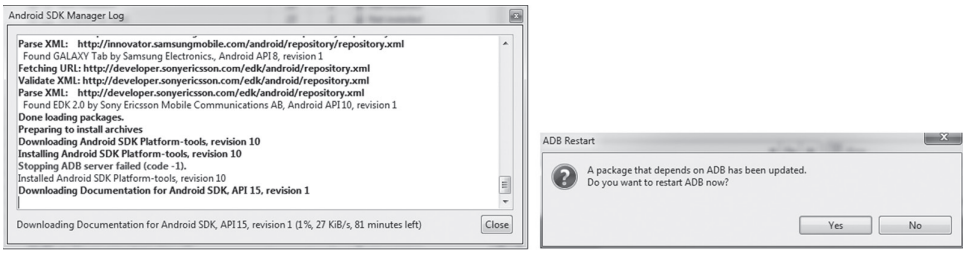


Figure A.7 The dialog box to accept the license terms for the selected packages and to begin installation.



(a) (b)

Figure A.8 (a) The Android SDK Manager Log showing the status of different packages. (b) The `ADB Restart` dialog box, prompting you to Restart ADB.

restart the Android Debug Bridge (ADB), as shown in Figure A.8b. Select the `Yes` button in the dialog to restart ADB.

The next dialog box is the `Android SDK Manager`, as shown in Figure A.9. The dialog box confirms that the Android SDK platform tools, `Android 4.2 (API 17)`, and its components have been successfully installed on your machine.

You do not need the Android SDK Manager window for now, so you can go ahead and close it.

An Android application is a combination of several small components that includes Java files, Extensible Markup Language (XML) resource and layout files, manifest files, and much more. It will be very time-consuming to create all these components manually. So, you will be using the following two applications to help you:

- **Eclipse IDE**—An integrated development environment (IDE) that makes the task of creating Java applications quite easy. It provides a complete platform for developing Java applications with compiling, debugging, and testing support.
- **Android Development Tools (ADT) plug-in**—A plug-in that is added to the Eclipse IDE and automatically creates the necessary Android files so you can concentrate on the process of application development.

Before you begin the installation of Eclipse IDE, first set the path of the JDK that you installed, as it will be required for compiling the applications. To set the JDK path on Windows,

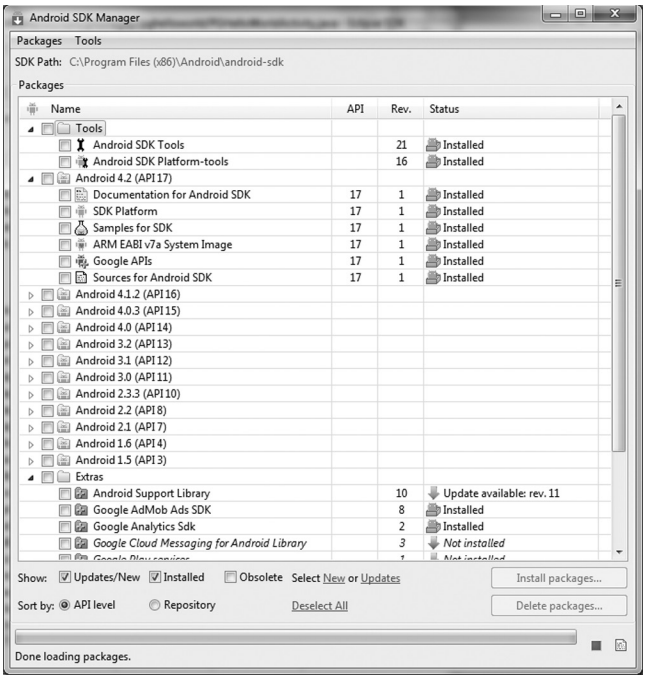


Figure A.9 The Android SDK Manager showing that all the desired packages have been successfully installed on the machine.

right-click on the `My Computer` icon and select the `Properties` option. From the `System Properties` dialog box that appears, select the `Advanced` tab, followed by the `Environment Variables` button. A dialog box, `Environment Variables`, pops up. In the `System variables` section, double-click on the `Path` variable. Add the full path of the JDK (`C:\Program Files\Java\jdk1.7.0_09\bin\java.exe`) to the `path` variable and select `OK` to close the windows.

Installing Eclipse

Eclipse IDE is a multilanguage software development platform that is commonly used for developing Java applications. You can add plug-ins to extend its features for developing applications in other languages. Eclipse can be downloaded from the following URL: `http://www.eclipse.org/downloads/`. `Eclipse Classic` and `Eclipse IDE for Java Developers` are recommended. Just remember that both the JDK and Eclipse must be for the same version, either 32 bit or 64 bit. The latest version, Eclipse Classic 4.2.1, is available at the time of this writing.

Eclipse is a self-contained executable file; that is, all you need to do to install Eclipse is to unzip the downloaded file to any desired folder. To launch Eclipse, run the `Eclipse.exe` file. Eclipse IDE starts by displaying its logo, followed by a `Workspace Launcher` dialog box, as shown in Figure A.10. The `Workspace Launcher` dialog prompts for the location of the workspace folder where the Eclipse project files will be stored. A default location is displayed that you can change by selecting the `Browse` button.

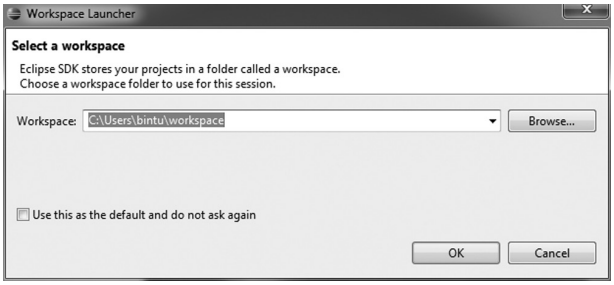


Figure A.10 The first screen you see after launching Eclipse IDE, asking for the workspace location to save applications.

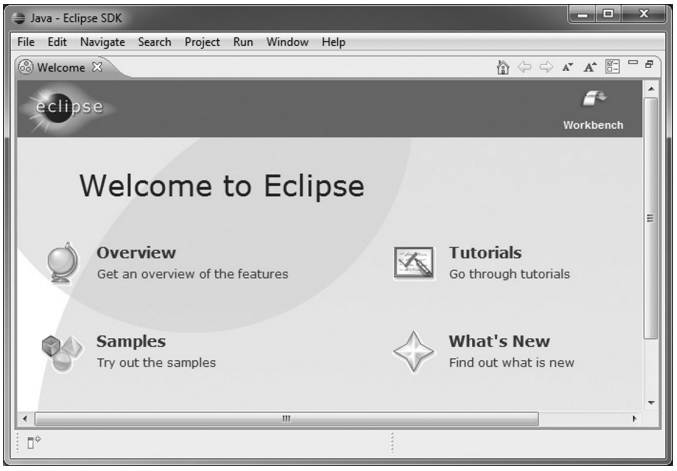


Figure A.11 The Eclipse welcome screen.

The checkbox `Use this as the default and do not ask again` can be checked if you do not want Eclipse to prompt for the workspace every time it is launched. Select the `OK` button to continue. When Eclipse finishes loading, an Eclipse welcome screen is displayed, as shown in Figure A.11.

Select the curved-arrow icon at the top right of the screen to go to the Workbench, as shown in Figure A.12.

You can see that all the windows in the Workbench (Package Explorer, Editor window, Debug window, and Task List) are blank at the moment. These windows will update their content as you develop PhoneGap applications for Android. One more step is needed before you begin the Android application development—installing the ADT plug-in.

Installing the Android Developer Tool (ADT) Plug-In

ADT is a plug-in for the Eclipse IDE that provides a powerful, integrated environment to build Android applications. It makes the task of developing Android applications quite easy. It integrates with Eclipse to add functionality for creating, testing, debugging, and deploying Android applications.

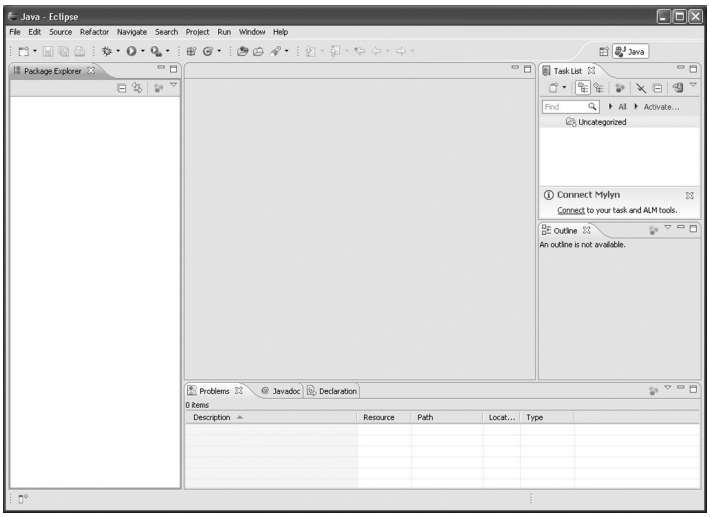
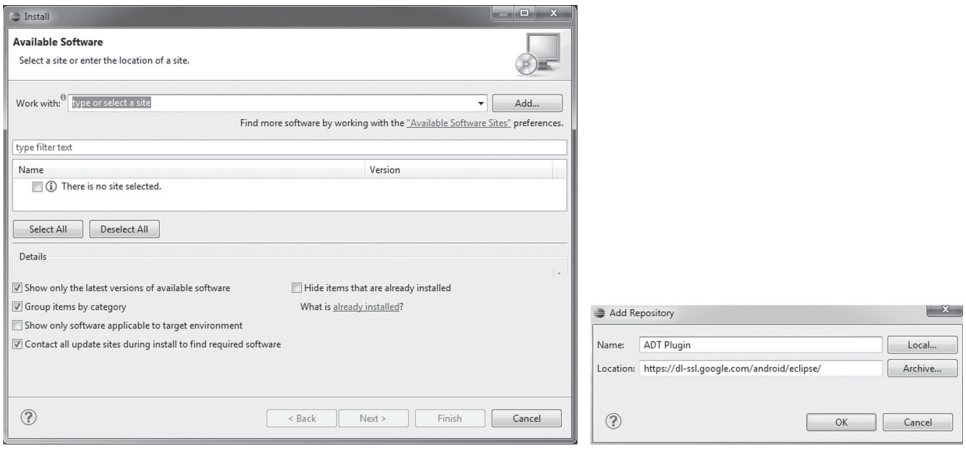


Figure A.12 The Eclipse Workbench, showing windows and panels.

To install the ADT plug-in, select the `Help->Install New Software...` option from the Eclipse IDE. You will see a dialog box asking for the location of the Web site from which you wish to install new software, as shown in Figure A.13a. Select the `Add` button to add a Web site or repository location. An `Add Repository` dialog box will appear, as shown in Figure A.13b. Enter the name of the repository in the `Name:` text box. Let us specify the name of the repository as `ADT Plugin`, although it can be any other address. In the `Location` box, specify the location of the repository as `https://dl-ssl.google.com/android/eclipse/`, followed by the `OK` button.

Eclipse will access the list of developer tools available at the specified site and display it, as shown in Figure A.14. In the figure, you can see that an entry named `Developer Tools` is



(a) (b)

Figure A.13 (a) The dialog box prompting for the location of the software installation Web site. (b) The dialog box to add the new repository information.

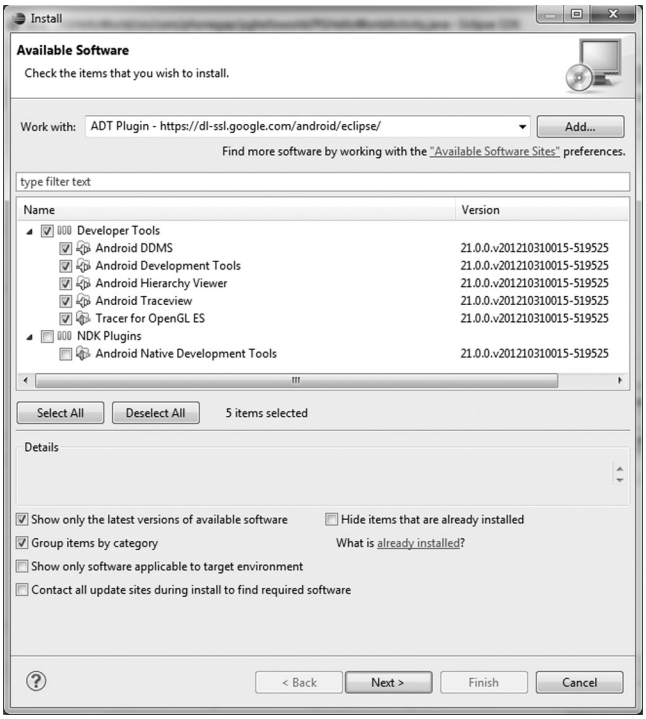


Figure A.14 The dialog box displaying the list of developer tools available in the added repository.

displayed, along with five child nodes: `Android DDMS`, `Android Development Tools`, `Android Hierarchy Viewer`, `Android Traceview`, and `Tracer for OpenGL ES`. We need to install all five tools, so select the parent node, `Developer Tools` (its child nodes will be autoselected), and select the `Next` button.

You will see a dialog box to review licenses for the ADT. Read the license agreement, check the radio button `I accept the terms of the license agreement` if you agree with the terms and conditions, and select the `Finish` button. The ADT plug-in will be downloaded and installed in Eclipse. After installation of the ADT plug-in, you get a `Software Updates` dialog box asking to restart Eclipse. To make the ADT plug-in show up in the IDE, you need to restart Eclipse. Select the `Restart Now` button from the `Software Updates` dialog box to make the installation changes take effect.

Note: If you do not want to take the time to install Eclipse, Android SDK Tools, and ADT plug-in, an ADT bundle is provided at: `http://developer.android.com/sdk/index.html`. Download the bundle and it will install all the essential Android SDK components and a version of the Eclipse IDE with built-in ADT for you.

Making the ADT Plug-In Functional

To make the ADT plug-in functional inside of Eclipse, the plug-in needs to point to the Android SDK. Launch the Eclipse IDE and select the `Window->Preferences` option. In the

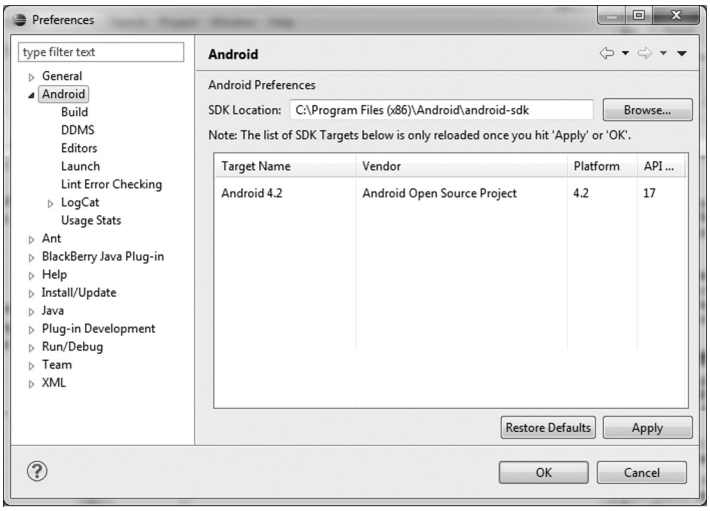


Figure A.15 The `Preference` window to specify the location of the Android SDK installation and the list of supportable platforms displayed after specifying the location of the Android SDK installation.

`Preferences` dialog box, select the Android node and set the `SDK Location` box to specify the path where Android SDK is installed on your disk (see Figure A.15).

Upon specifying the path of the Android SDK, a list of SDK targets will be displayed. You can now develop and test the Android applications against any of the displayed targets. Select the `Apply` button, followed by the `OK` button, to reload the SDK targets and close the `Preferences` window. Eclipse now has the ADT plug-ins attached. That is, you can now develop and run the PhoneGap application for Android.

Appendix B: Using Github

In this appendix, we will learn to:

- Create a Github account
- Create a local repository on our local machine
- Update the Github account with the local repository created on our machine

The first step is to create an account in Github. So, visit the `https://github.com/` URL to sign up for an account. Because we will be managing our Github repository through our Windows-based PC, let us download Github for Windows from `http://windows.github.com/`. Double-click the downloaded executable file and follow the wizard to install Github on our computer. Click the GitHub shortcut to launch it. The first screen that pops up is the login screen to the Github account. Enter the Github ID and password that we used when we created an account on the Github site (see Figure B.1a). Upon successful login, we get a screen as shown

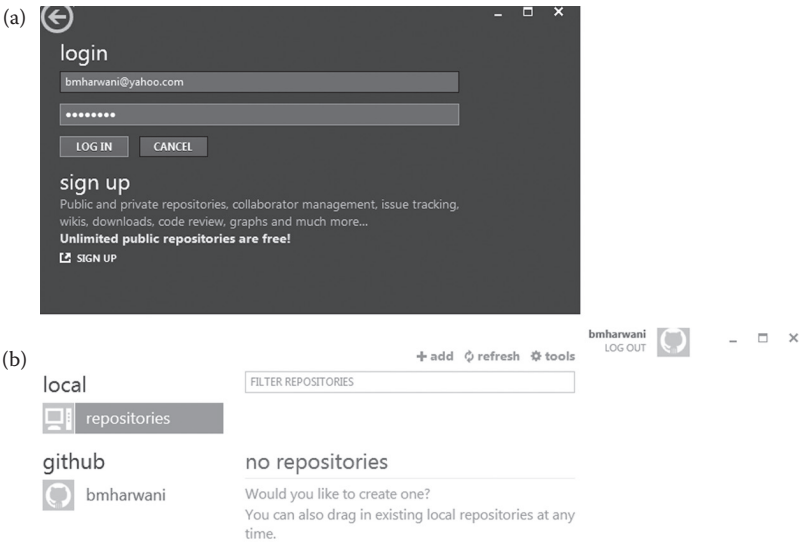


Figure B.1 (a) Logging into a Github account. (b) The page showing information of the local and Github repository.

in Figure B.1b. We can see that the figure indicates that there is neither any local repository nor any repository on Github's account. So, let us create a local repository by clicking the `+add` link shown on the top.

We get a page that asks for the information for the new local repository. We enter the title and a small description of the new repository. The page will show the location on our disk drive where the newly created local repository will be stored. Check the `Push to github` checkbox because we will be pushing or uploading the local repository into our Github account later. The Github account to which the local repository is associated will also be displayed. Figure B.2a shows that the local repository will be pushed to the `bmharwani` account on Github. After entering the information of the new local repository, click the `Create` button to create the local repository at the specified location on the disk. The newly created local repository appears in the `local` section. The name `bmharwani/PhoneGapApps` that is assigned to the local repository indicates that `PhoneGapApps` is the repository created for the `bmharwani` account (see Figure B.2b).

In the repository titled `bmharwani/PhoneGapApps`, when we hover the mouse over the right arrow, the following text will appear, `open this repo`. Click on the right arrow to open the newly created local repository, `PhoneGapApps`. Upon opening the local repository, we find that two files, `.gitattributes` and `.gitignore`, are automatically created for us by default, as shown in Figure B.3.

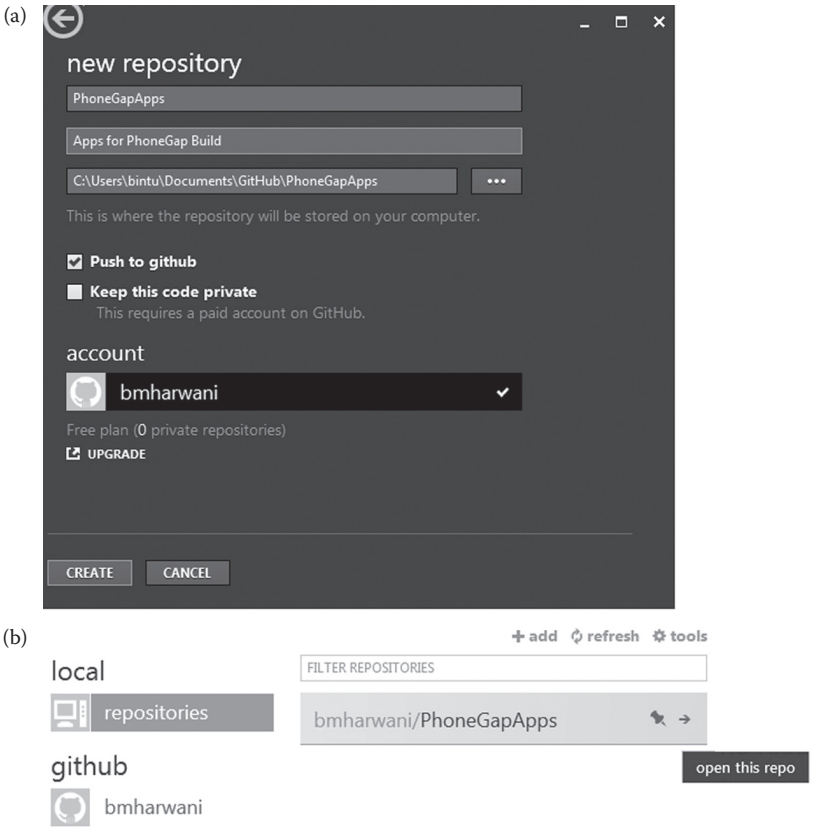


Figure B.2 (a) Entering information of a new local repository. (b) The newly created local repository appears in the local section.

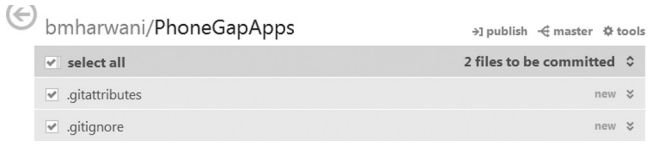


Figure B.3 The new repository with two default files, `.gitattributes` and `.gitignore`.

To copy files of our application into the local repository, we need to open the repository in the file explorer. So, select the `tools` icon at the top and select the `open in explorer` option from the drop-down menu that pops up. To the repository folder, copy the files of the application that we want to upload on the PhoneGap Build service, as shown in Figure B.4a. The figure shows that we copied three files in the local repository: `index.html`, `config.xml`, and `icon.png`. The three application files copied into the repository folder will then appear in our local repository, `bmharwani/PhoneGapApps`, as shown in Figure B.4b. All the files appear to be in the `uncommitted` state. In order to save the changes, the files need to be committed. In the dialog on the right side, we need to enter a title or small description to identify the commit action and click the `COMMIT` button at the bottom of the dialog. The title that is assigned to the commit action is used to identify it and cancel the commit operation, if required.

After clicking the `COMMIT` button, there is one more step to make the changes made to the local repository permanent. The commits made at the moment are entitled `unsynced commits`. Below the heading `unsynced commits`, we will see text showing some ID along with the text `Saving the three files`, as shown in Figure B.5. Click the text `Saving`

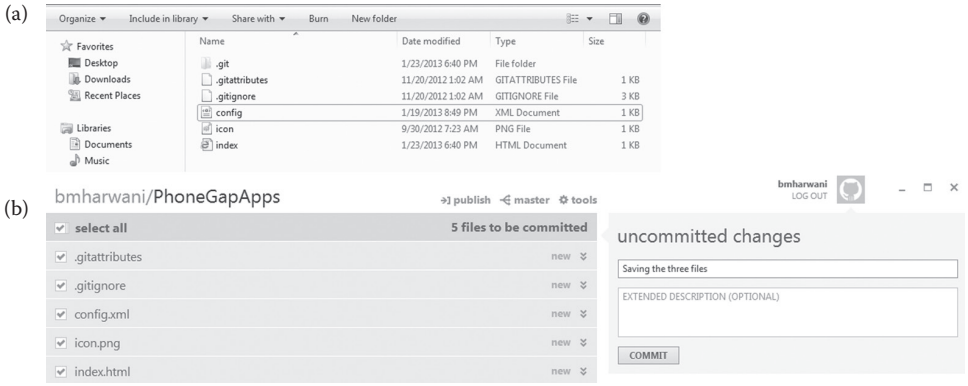


Figure B.4 (a) Copying the files of an app into the local repository using file explorer. (b) The copied files appear in the local repository with "uncommitted changes."

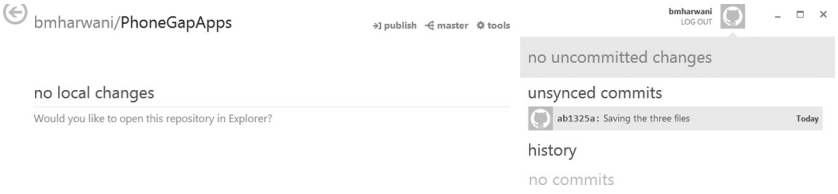


Figure B.5 The page informing us that the commits made are in the "unsynced commits" state.

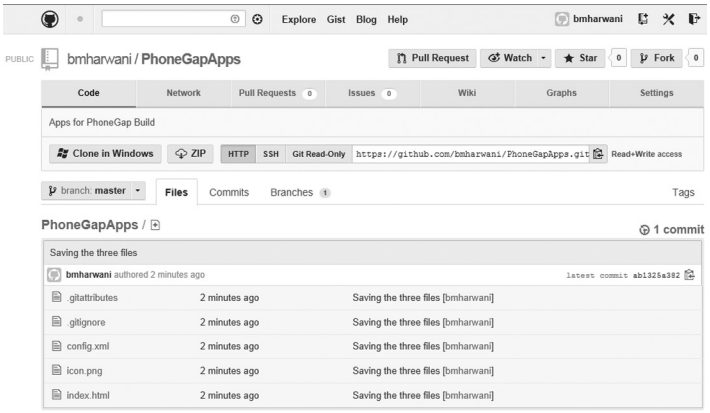


Figure B.6 The committed files of the application appear in the Github account.

`the three files` to save the files that we added through the file explorer into the local repository.

To view the saved local repository files in the Github account, click the `tools` icon at the top and select the option `view on github` from the drop-down menu that pops up. The Github account will open, showing the files that we created in our local repository, `PhoneGapApps`, as shown in Figure B.6.

Now, our Git repository is ready by name, `bmharwani/PhoneGapApps`, to be uploaded on the PhoneGap Build service.

Information Technology / Programming Languages

PhoneGap is a standards-based, open-source development framework that can be deployed to any mobile device without losing the features of the native app—allowing for access to device contacts, the local file system, camera, and media on multiple platforms without requiring users to write a single line of code.

Ideal for intermediate to advanced users, **PhoneGap Build: Developing Cross Platform Mobile Applications in the Cloud** offers the comprehensive coverage you need to harness the power of this dynamic tool. It provides complete coverage of the cloud computing platform and the theories behind cloud computing, using a series of engaging examples.

The book explains the differences between existing mobile platforms, the different types of browsers they support, and the programming languages and integrated development environment required to develop apps for each of them. It then describes how PhoneGap makes the task of developing cross platform mobile apps easier. This book will teach you how to use:

- HTML5, CSS3, and JavaScript to develop apps for devices across various mobile operating systems
- PhoneGap Build to develop mobile apps in the cloud
- PhoneGap with Sencha Touch and jQuery Mobile
- Back-end databases to store and retrieve information

The text starts with simpler applications and gradually moves toward describing advanced concepts and how to exploit different application programming interfaces and methods. By the time you finish the book, you will learn how to develop feature-rich mobile applications that can run on the cloud to support different platforms.

Supplying authoritative guidance and proven best practices for designing cloud-based applications, the book is an ideal reference for cloud system developers, architects, and IT professionals. It is also suitable for use in instructional settings.

CRC Press
Taylor & Francis Group
an **informa** business
www.crcpress.com

6000 Broken Sound Parkway, NW
Suite 300, Boca Raton, FL 33487
711 Third Avenue
New York, NY 10017
2 Park Square, Milton Park
Abingdon, Oxon OX14 4RN, UK

K20423
ISBN: 978-1-4665-8974-2
90000
9 781466 589742

www.auerbach-publications.com