File System, Storage, and Local Database for Hybrid Mobile Application Development

Introduction

- With the ever increasing storage capacities on offer with each mobile device, whether built-in storage or available as an expansion through a card, developers have the ability to interact with and manipulate files stored on the device as well as utilize API functionality to cache content.
  - Saving a file to device storage
  - Opening a local file from device storage
  - Displaying the contents of a directory
  - Creating a local SQLite database
  - Uploading a file to a remote server via a POST request
  - Caching content using the web storage local storage API

Saving a file to device storage

- We will allow the user to enter a remote URL for a file into a textbox, and download and save that file to their mobile device.
- The steps to be performed are as follows:

1. Create the initial HTML layout for our application. We're going to use the XUI JavaScript library to easily access DOM elements, so we'll include the reference to the file within the head tag along with the cordova-2.0.0.js file.
2. Below the Cordova JavaScript reference let's also create a new JavaScript tag block to hold our custom code.

```
<!DOCTYPE html>
<html>
<head>
        <meta name="viewport" content="user-scalable=no, initial-scale=1,
        maximum-scale=1, minimum-scale=1, width=device-width;" />
        <title>File Download</title>
        <script type="text/javascript" src="xui.js"></script>
        <script type="text/javascript“ src="cordova-2.0.0.js"></script>
        <script type="text/javascript">
        </script>
</head>
<body>   </body>
</html>
```

Saving a file to device storage

3. Within the body tag create two input elements. Set the first element type attribute to text and set the id attribute to file_url.
4. Set the second input element type attribute to button, the id attribute to download_btn and the value to equal Download.
5. Finally include a new div element and set the id attribute to message. This will be the container into which our returned output is displayed. This is shown in the following code:

```
<body>
        <input type="text"
                id="file_url" value="" />
        <input type="button"
                id="download_btn" value="Download" />
        <div id="message"></div>
</body>
```

Saving a file to device storage

6. Within the empty JavaScript tag block we need to define a global variable called **downloadDirectory** that will reference the location on the device to store the retrieved file. We'll also add this variable in our event listener for our application which will run once the native PhoneGap code has been loaded:

```
var downloadDirectory;
document.addEventListener("deviceready", onDeviceReady, true);
```

7. We can now write our **onDeviceReady** function. The first thing we need to do is to access the root filesystem on the device. Here we are requesting access to the persistent file system. Once a reference has been established we run the **onFileSystemSuccess** method to continue.

Saving a file to device storage

8. We are then binding a click function to the download_btn element using XUI, which will run the download function when clicked:

```
function onDeviceReady() {
        window.requestFileSystem(LocalFileSystem.PERSISTENT, 0,
        onFileSystemSuccess, null );
        x$('#download_btn').on( 'click', function(e) {
                download();
        });
}
```

9. With the connection made to the device storage, we can reference the root system using the **fileSystem** object provided by PhoneGap. Here we then call the **getDirectory** method, providing the name of the directory to gain access to. If it doesn't exist it will be created for us. After a successful **response**, the returned **DirectoryEntry** object is assigned to the **downloadDirectory** variable we set earlier

Saving a file to device storage

```
function onFileSystemSuccess(fileSystem) {
        fileSystem.root.getDirectory('my_downloads‘,
                {create:true},
                function(dir) {
                        downloadDirectory = dir;
                },fail);
}
```

10. Our **download** function will be run when the user clicks the **Download** button. We first need to obtain the URL provided by the user from the text input box. We can pass the value through, to a new custom method called **getFileName**, which will split the string and return the filename and extension for use later in the function. We can now set a user-friendly message into the message container to inform them of our progress.
11. Next we instantiate a new **FileTransfer** object from the PhoneGap API to assist us in downloading the remote object. The download method accepts the remote URL to download, the directory on the device to save the file, and the success and error callback functions. After a successful operation, we will inform the user of the local path where the file was saved:

Saving a file to device storage

```
function download() {
var fileURL = document.getElementById('file_url').value;
        var localFileName = getFilename(fileURL);
        x$('#message').html('Downloading ' + localFileName);
        var fileTransfer = new FileTransfer();
        fileTransfer.download( fileURL, downloadDirectory.fullPath + '/' +
                localFileName, function(entry){
                        x$('#message').html('Download complete. File saved
                        to: ' + entry.fullPath);
                },
        function(error){
                alert("Download error source " + JSON.stringify(error));
        }
        );
}
```

HMAD

Saving a file to device storage

12. Include the custom function to obtain the filename and extension of the remote file:

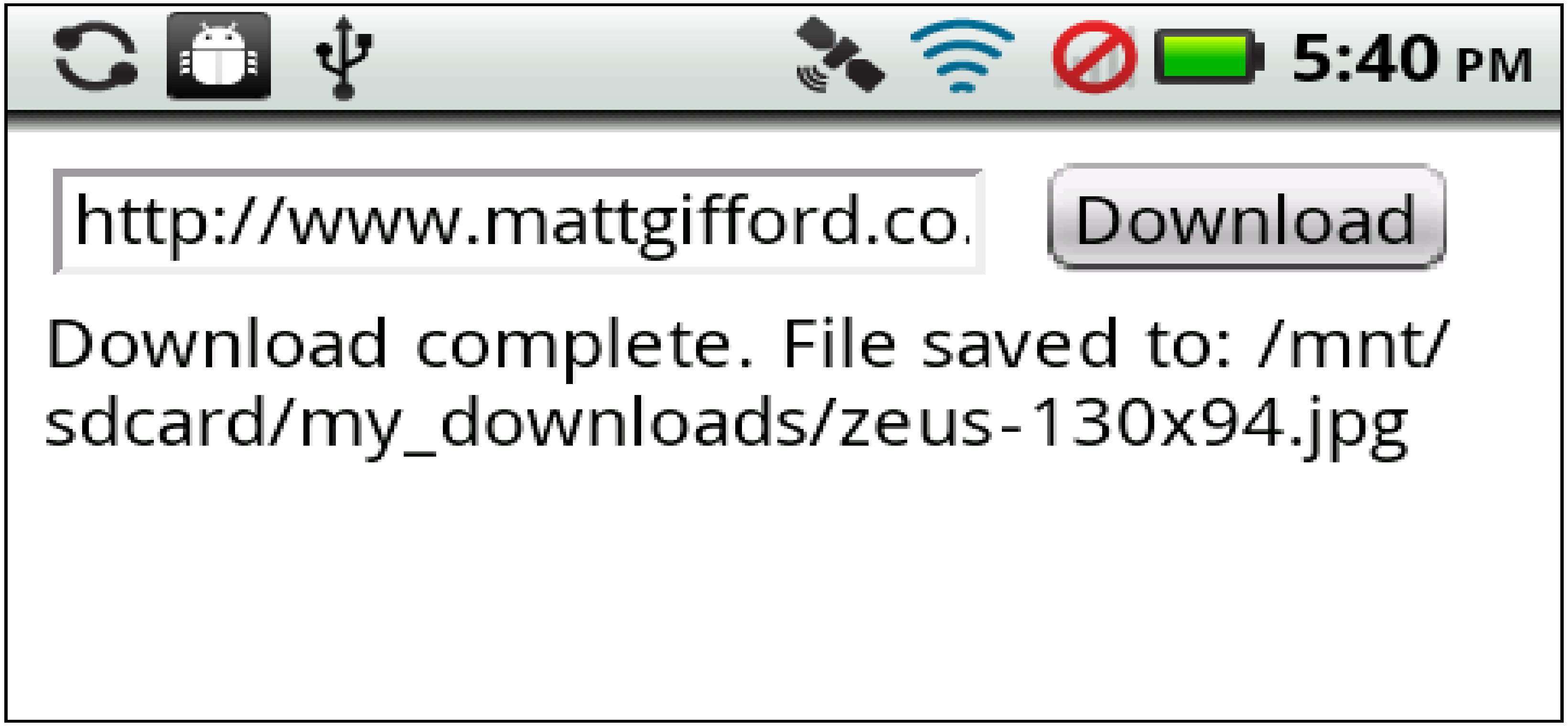
```
function getFilename(url) {
        if (url) {
                var m = url.toString().match(/.*\/(.+?)\./);
                if (m && m.length > 1) {
                        return m[1] + '.' + url.split('.').pop();
                }
        }
        return "";
}
```

13. Finally, we supply the fail method, which is the generic error handler for all of our functions within the application.

```
function fail(error) {
        $('#message').html('We encountered a problem: ' + error.code);
}
```

Saving a file to device storage

14. When we run the application, we can specify a remote file to download to the local storage and provide the file's location on the device. The result would look something like the following screenshot:



Download complete. File saved to: /mnt/sdcard/my_downloads/zeus-130x94.jpg

Opening a local file from device storage

- We will build an application that will create a text file on the phone's storage file system, write content into the file, and then open the file to display the content, as listed in the following steps.

1. Create the initial HTML layout for our application. We're going to use the XUI JavaScript library to easily access DOM elements, so we'll include the reference to the file within the head tag along with the cordova-2.0.0.js file.

2. Below the Cordova JavaScript reference let's also create a new JavaScript tag block to hold our custom code. This is shown in the following code:

Opening a local file from device storage

```
<!DOCTYPE html>
<html>
<head>
        <meta name="viewport" content="user-scalable=no,initial-scale=1,
        maximum-scale=1, minimum-scale=1, width=device-width;" />
        <title>Open File</title>
        <script type="text/javascript" src="xui.js"></script>
        <script type="text/javascript“ src="cordova-2.0.0.js"></script>
        <script type="text/javascript">

        </script>
</head>
<body>

</body>
</html>
```

Opening a local file from device storage

3. Within the body tag create two input elements. Set the first element type attribute to text and set the id attribute to my_text.

4. Set the second input element type attribute to button, the id attribute to savefile_btn, and the value to equal Save.

5. Finally include two new div elements. Set the first element id attribute to message. This will be the container into which our returned output is displayed. Set the second element id attribute to contents. This will display the contents of the file:

```
<body>
        <input type="text" id="my_text" />
        <input type="button" id="saveFile_btn“
        value="Save" />
        <div id="message"></div>
        <div id="contents"></div>
</body>
```

Opening a local file from device storage

6. Within the empty JavaScript tag block we need to define a global variable called **fileObject** that will reference the file on the device. We'll also add in our event listener for our application that will run once the native PhoneGap code has been loaded. The **onDeviceReady** method requests access to the persistent filesystem root on the device. Once obtained, it will execute the **onFileSystemSuccess** method, shown as follows:

```
var fileObject;
document.addEventListener("deviceready", onDeviceReady, true);
function onDeviceReady() {
        window.requestFileSystem(
                LocalFileSystem.PERSISTENT, 0,
                onFileSystemSuccess, fail);
}
```

Opening a local file from device storage

7. With the connection made to the device storage, we can reference the root system using the **fileSystem** object provided by PhoneGap. Here we then call the **getFile** method, providing the name of the file we wish to open. If it doesn't exist it will be created for us.

```
function onFileSystemSuccess(fileSystem) {
        fileSystem.root.getFile("readme.txt",
                {create: true, exclusive: false},
                gotFileEntry, fail);
}
```

8. After a successful response, the returned **FileEntry** object is assigned to the **fileObject** variable we created earlier. At this point we can also bind a click handler to our save button, which will run the **saveFileContent** function when clicked.

```
function gotFileEntry(fileEntry) {
        fileObject = fileEntry;
        x$('#saveFile_btn').on('click', function() {
                saveFileContent();
        });
}
```

Opening a local file from device storage

9. As we're dealing with a local file that, as of yet, doesn't have any content, we can use methods within the PhoneGap API to write to the file. The **saveFileContent** method will access the **fileObject** and call the **createWriter** method to start this process.

```
function saveFileContent() {
        fileObject.createWriter(gotFileWriter, fail);
}
```

10. Let's now create the **gotFileWriter** method called as a callback from the save function. We'll send the value from the **my_text** input field into the **writer. write()** method to populate the file content. After the writing has finished, we'll output a status message into the message **div** element and then instantiate the **FileReader** object to read the file contents.

11. We will then pass the **fileObject** into the **reader.readAsText()** method to return the text content of the file. After the read has completed, we will output the contents into the **div** element for display.

Opening a local file from device storage

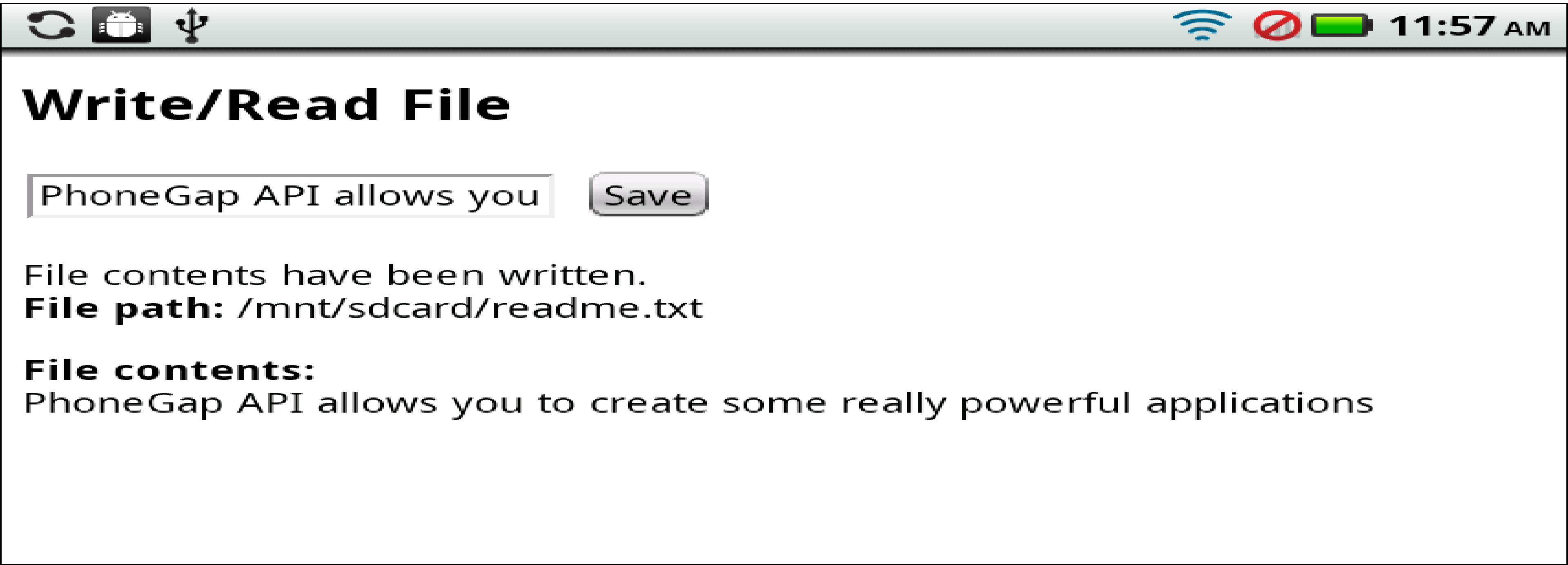
```
function gotFileWriter(writer) {
        var myText = document.getElementById('my_text').value;
                writer.write(myText);
        writer.onwriteend = function(evt) {
                x$('#message').html('<p>File contents have been written.<br
                /><strong>File path:</strong> ' + fileObject.fullPath + '</p>');
                var reader = new FileReader();
                reader.readAsText(fileObject);
                reader.onload = function(evt) {
                        x$('#contents').html('<strong>File
                        contents:</strong> <br />' + evt.target.result);
                };
        };
}
```

Opening a local file from device storage

12. Finally, include the fail error handler method to catch any problems or errors:

```
function fail(error) {
        alert(error.code);
}
```

13. When we run the application on our device, we should see output similar to the following screenshot:



File contents have been written.
File path: /mnt/sdcard/readme.txt

File contents:
PhoneGap API allows you to create some really powerful applications

Creating a local SQLite database

- We will create a mobile application that will allow us to store text entries into a local SQLite database and then query the database to retrieve all saved items.

1. Firstly, create the HTML layout for our page, including the reference to the cordova-2.0.0.js file:

```
<!DOCTYPE html>
<html>
<head>
        <meta name="viewport" content="user-scalable=no, initial-scale=1,
        maximum-scale=1, minimum-scale=1, width=device-width;" />
        <title>My ToDo List</title>
        <script type="text/javascript“ src="cordova-2.0.0.js"></script>

</head>
<body>
        <h1>My ToDo List</h1>

</body>
</html>
```

Creating a local SQLite database

2. In this example we will be referencing certain elements within the DOM by class name. For this we will use the XUI JavaScript library. Add the **script** reference within the **head tag** of the document to include this library.
3. Below the PhoneGap JavaScript include, "write a new JavaScript tag block", and within this define an **onDeviceReady** event listener to ensure the device is ready and fully loaded before the application proceeds to execute the code.

```
<title>My ToDo List</title>
<script type="text/javascript" src="cordova-2.0.0.js"></script>

<script type="text/javascript" src="xui.js"></script>
<script type="text/javascript">
// PhoneGap code here
</script>
```

4. Add an input element within the body tags with the **id** attribute set to **list_action**. This will allow the user to add entries into the database list.

Creating a local SQLite database

5. Below the input element, add a button element with the **id** attribute set to **saveItem**.
6. Let's also add two **div** elements to hold any generated data. The first, with the **id** attribute set to **message**, will hold any database connection error messages if we have issues trying to connect. The second, with the **id** attribute set to **listItems**, will act as a container into which our generated list will be placed for display.

```
<body>
<h1>My ToDo List</h1>

    <input type="text"       id="list_action" />

    <input type="button"     id="saveItem" value="Save" />

    <div id="message"></div>

    <div id="listItems"></div>

</body>
```

Creating a local SQLite database

7. With the layout complete, let's move on to adding our custom JavaScript code. To begin with, we need to define the **deviceready** event listener. As we are using the XUI library, we will write this function as follows:

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



});
```

8. As we want to set the inner HTML values for the **list** and **message div** containers, let's define the references to those particular elements. We'll also create a global variable called **db** that will eventually hold our database connection.
9. Now bind a click handler to the **saveItem** button element. When pressed, it will run the **insertItem** method to add a new record to the database.
10. We now need to create a reference to our SQLite database. The PhoneGap API includes a function called **openDatabase** that creates a new database instance or opens the database if it already exists. The returned object will allow us to perform transactions against the database.

Creating a local SQLite database

```
var listElement = x$('#listItems');
var messageElement = x$('#message');
var db;
x$('#saveItem').on('click', function(e) {
insertItem();
});

// Create a reference to the database
function getDatabase() {
return window.openDatabase("todoListDB“,"1.0", "ToDoList Database",
        200000);
}
```

11. We can now include the call to and create the **onDeviceReady** method. Here we assign the database instance to the variable **db**, which will allow us to perform a transaction into the database. In this case, we'll execute a simple SQL script to create a table called **MYLIST** if it doesn't already exist.

Creating a local SQLite database

```
/ Run the onDeviceReady method
onDeviceReady();

// PhoneGap is ready
function onDeviceReady() {
db = getDatabase();
db.transaction(function(tx) {
tx.executeSql('CREATE TABLE IF NOT EXISTS MYLIST
(id INTEGER PRIMARY KEY AUTOINCREMENT, list_action)');
}, databaseError, getItems);
}
```

12. Now define the **getItems** method, which is run on a successful callback from the database transaction in the previous method. Once more we reference the database object and perform another transaction, this time to select all items from the table.

Creating a local SQLite database

```
// Run a select statement to pull out all records
function getItems() {
db.transaction(function(tx) {
tx.executeSql('SELECT * FROM MYLIST', [], querySuccess, databaseError);
   }, databaseError);
}
```

13. Having received the results from the select query, we can loop over the results and create list item elements that we can then set within the **list div** container. Here we can reference the **id** and **list_action** columns from the results, drawn from the SQLite database table we created earlier. We'll display the total number of records stored.

Creating a local SQLite database

```
// Process the SQLResultSetList
function querySuccess(tx, results) {
var len = results.rows.length;
var output = '';
for (var i=0; i<len; i++){
        output = output +
        '<li id="' + results.rows.item(i).id + '">' +
        results.rows.item(i).list_action + '</li>';
        }
messageElement.html('<p>There are ' + len + ' items in your list:</p>');
listElement.html('<ul>' + output + '</ul>');
}
```

14. We initially bound a click event handler to our **saveItem** button. Let's now create the **insertItem** method that the click handler would invoke. We want to take the value of the **list_action** text input box and pass that into the database transaction when we run an insert query. A successful insert will call our **getItems** method to query the database and populate the list with all updated information from our database.

Creating a local SQLite database

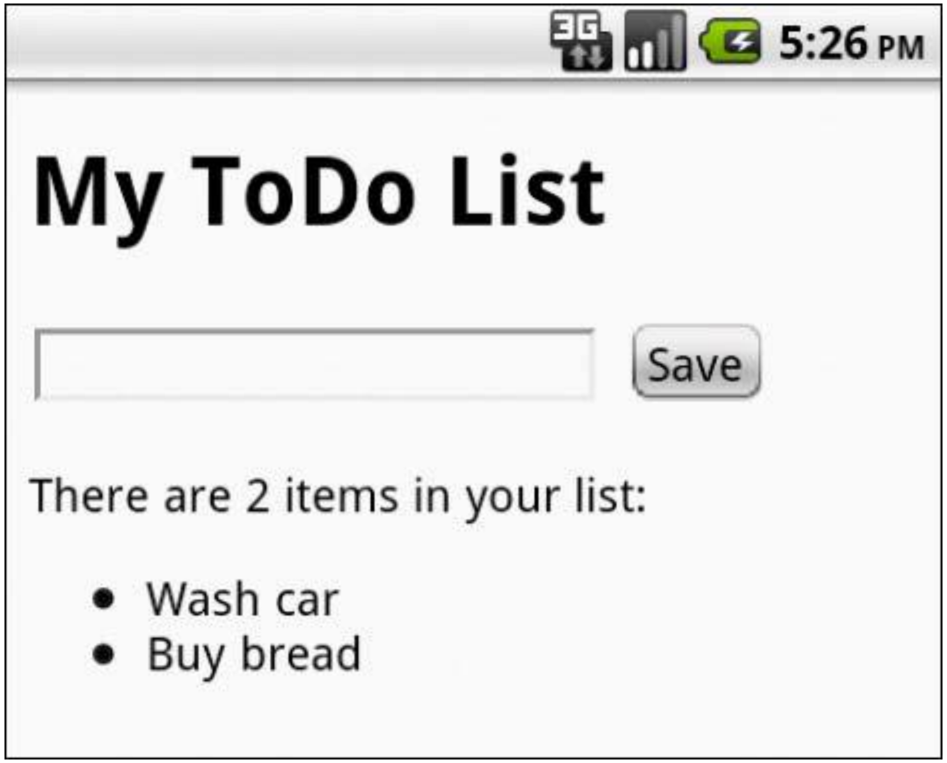
```
// Insert a record into the database
function insertItem() {
var insertValue =
document.getElementById('list_action').value;
db.transaction(function(tx) {
tx.executeSql('INSERT INTO MYLIST (list_action) VALUES ("' + insertValue + '")');
}, databaseError, getItems);

// Clear the value from the input box
document.getElementById('list_action').value = '';
}
```

15. Finally, let's include our **databaseError** fault handler method to display any issues we may encounter from the database and display them in the **message div** element.

```
// Database error handler
function databaseError(error) {
messageElement.html("SQL Error: " + error.code);
}
```

Creating a local SQLite database

16. When we run the application on the device, your output should look something like the following screenshot:



5:26 PM

My ToDo List

There are 2 items in your list:

- Wash car
- Buy bread

THANKS