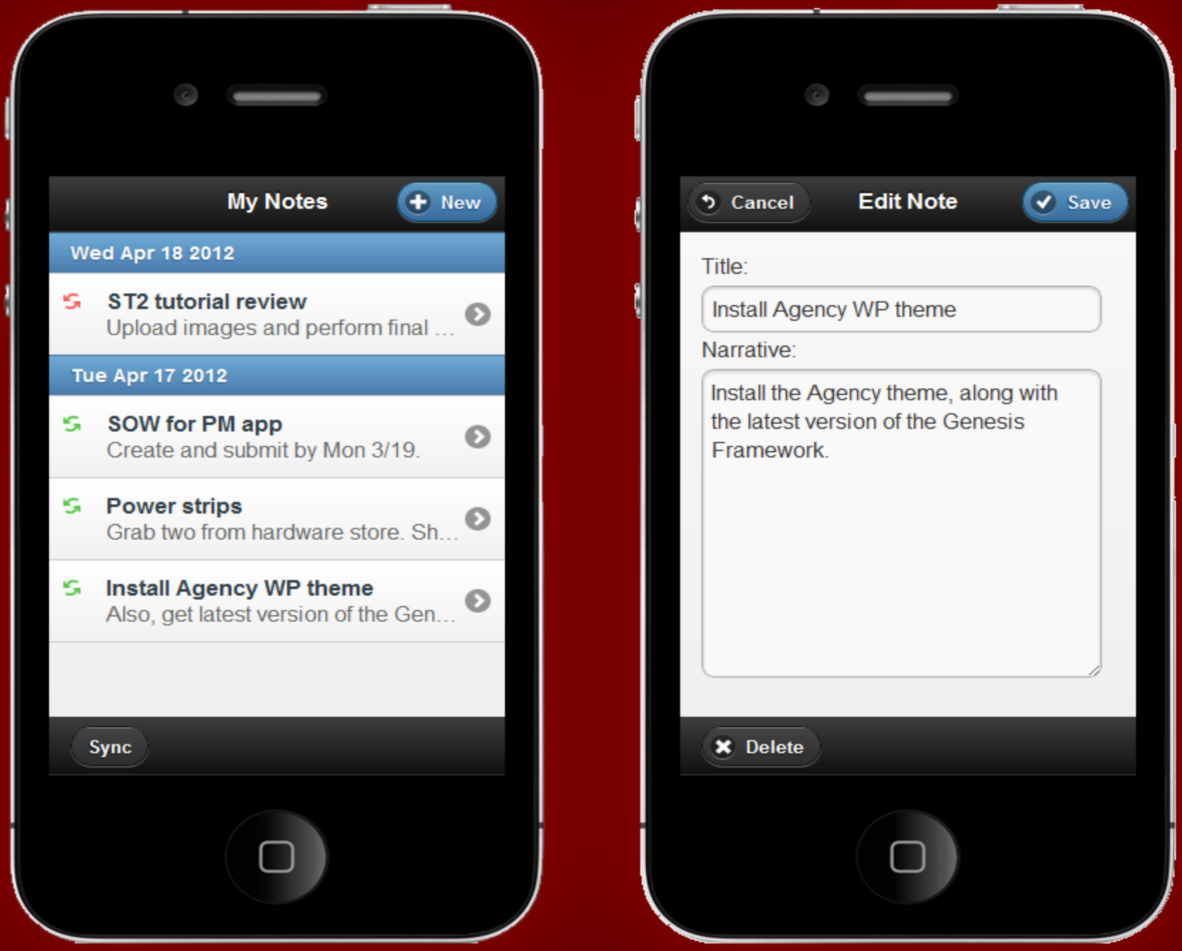
How to Build a jQuery Mobile Application


My Notes
New
Wed Apr 18 2012
ST2 tutorial review
Upload images and perform final …
Tue Apr 17 2012
SOW for PM app
Create and submit by Mon 3/19.
Power strips
Grab two from hardware store. Sh…
Install Agency WP theme
Also, get latest version of the Gen…
Sync
Cancel
Edit Note
Save
Title:
Install Agency WP theme
Narrative:
Install the Agency theme, along with the latest version of the Genesis Framework.
Delete

A hands-on guide to mastering the essential elements of a jQuery Mobile application

Jorge Ramon

How to build a jQuery Mobile Application

A hands-on guide to mastering the essential elements of a jQuery Mobile application

Book Version: 1.1

www.miamicoder.com

About The Book

Goals

How to Build a jQuery Mobile Application will guide you, step by step, through the process of building a mobile application using jQuery Mobile. The book's hands-on approach will allow you to learn the following topics:

- The building blocks of a jQuery Mobile application.
- How to create a user interface with jQuery Mobile.
- How to render data using lists, and how to style list items.
- How to edit data using form elements.
- How to save application data on the device, across browser sessions.
- How to synchronize application data with a server.

Besides helping you learn jQuery Mobile in a short time, the book will give you great pointers, which you can use when developing your own applications.

Here is a detailed view of what's ahead of us:

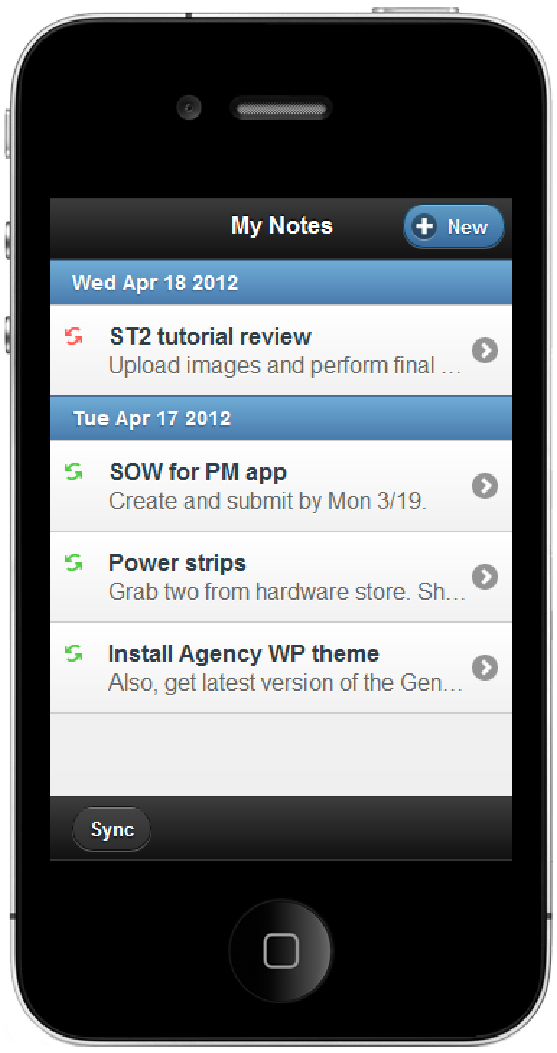
Table of Contents

About Warranties and Trademarks

Chapter 1: Introducing the Notes Application

What You Will Learn In this Book

In this book, you will learn how to create a jQuery Mobile application that allows its users to take notes, store them on the device running the app, and synchronize them with a server. We will call our app The Notes Application.



Wed Apr 18 2012

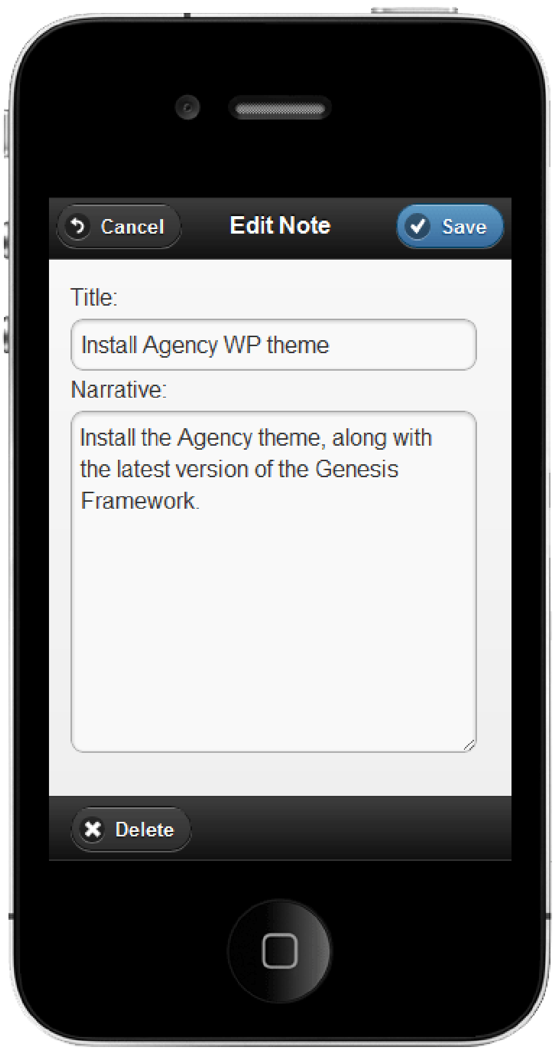
ST2 tutorial review
Upload images and perform final …

Tue Apr 17 2012

SOW for PM app
Create and submit by Mon 3/19.

Power strips
Grab two from hardware store. Sh…

Install Agency WP theme
Also, get latest version of the Gen…



Title: Install Agency WP theme

Narrative: Install the Agency theme, along with the latest version of the Genesis Framework.

While building the Notes app we will dive into the following subjects:

- Building blocks of a jQuery Mobile application.
- Rendering data using list views.
- Editing data using form elements.
- Device-side data persistence across browser sessions.
- Synchronization of application data with a server.
- Navigation in a multi-view application.
- Behavior-driven development with the Jasmine Framework.

In this chapter, we are going to talk about the overall design of the application, define its features, and design the user interface.

Let's get started.

The Development Approach

We will build the application following the Model-View-Controller (MVC) pattern. We will write Model and Controller logic using JavaScript modules and classes, and we will use jQuery Mobile to create our Views.

We are going to follow a behavior-driven approach to develop the business logic of the application. For each use case, we will first create a specification, and then we will implement the specification in the application.

Features of the Notes App

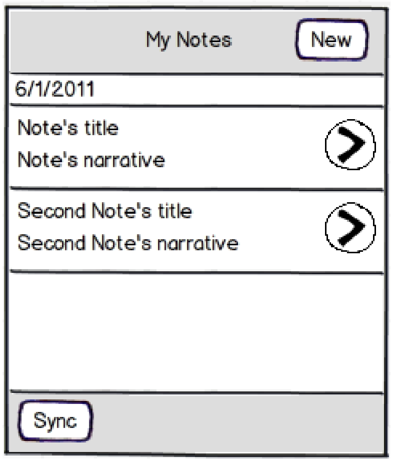
The Notes application has a simple feature set. We want to give our customers the following basic abilities:

- Create, edit, delete and view notes.
- Store notes on the mobile device running the app, across browser sessions.
- Synchronize notes with a server.

Designing the Notes List

The main view of the application will render the list of existing notes. We will call this view the Notes List, and it will be the first view our users see when they launch the application.

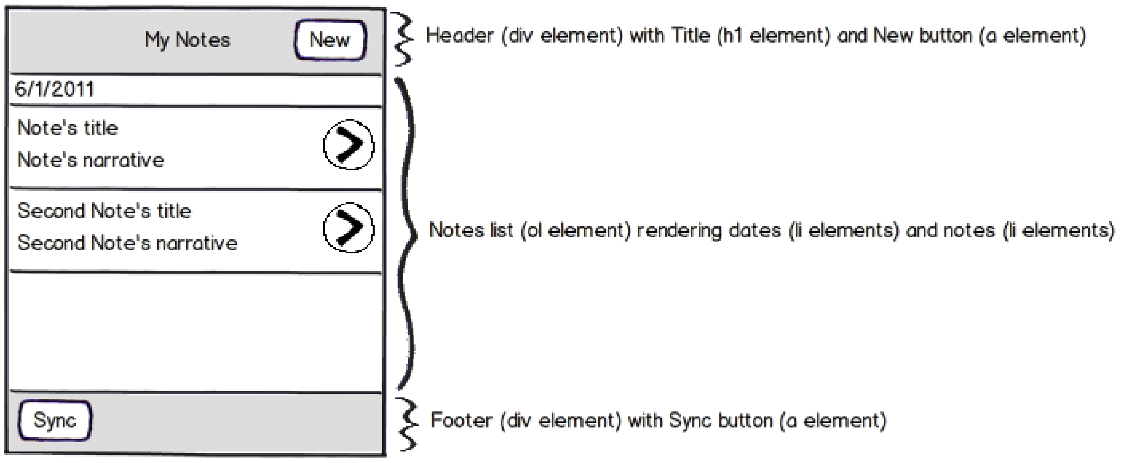
We are going to build the Notes List view so it looks similar to this mock-up:



As the mock-up indicates, the view will consist of the following parts:

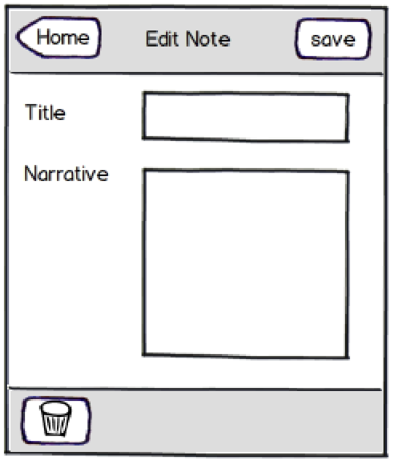
- A top toolbar containing the view's Title and the New button. The new button will connect with a second view, the Note Editor, which will allow our users to create new notes.
- A list view that will render the existing notes on the device. A tap on any element of the list will cause the selected note to be rendered in the Note Editor view.
- A bottom toolbar containing the Sync button. The Sync button will allow users to upload the existing notes to the server.

Here is a more descriptive mock-up, which depicts each of the view's parts and the html elements with which we will build them:


My Notes
New
6/1/2011
Note's title
Note's narrative
Second Note's title
Second Note's narrative
Sync
Header (div element) with Title (h1 element) and New button (a element)
Notes list (ol element) rendering dates (li elements) and notes (li elements)
Footer (div element) with Sync button (a element)

Designing the Note Editor

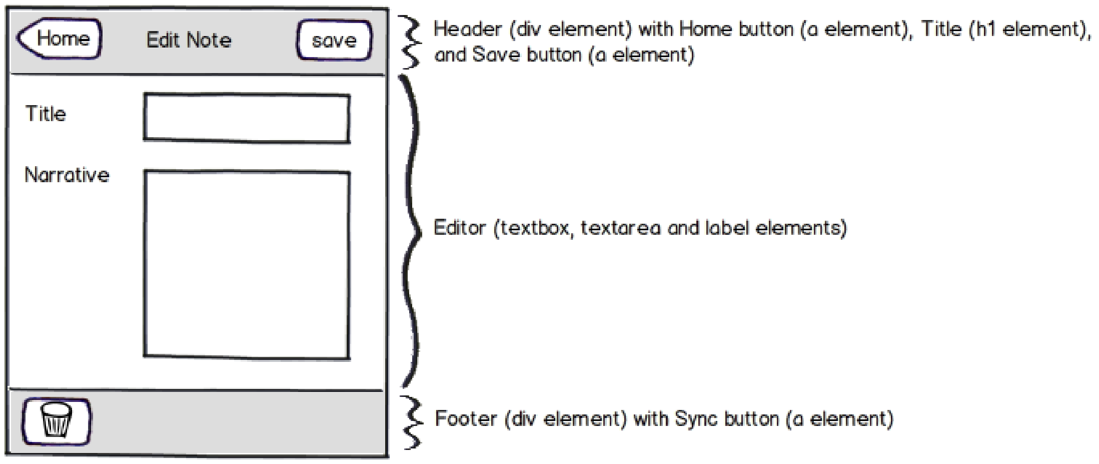
Of course, we also need an interface for our users to create, edit, and delete notes. We will name this view Note Editor, and it will look just like this mock-up:



The Note Editor view will consist of the following parts:

- A top toolbar containing the view's Title, and the Home and Save buttons. A tap on the Home button will cancel any changes made to the selected note, and trigger a transition back to the Notes List view. A tap on the Save button will cause the changes to the selected note to be saved, followed by a transition back to the Notes List.
- A form, containing fields for the note's title and narrative.
- A bottom toolbar containing the Delete button. Tapping this button will activate a small dialog that we will use to require confirmation from the user before deleting the note.

Here are each of the view's parts and the html elements we will use to build them:


Home
Edit Note
save
Title
Narrative
Header (div element) with Home button (a element), Title (h1 element), and Save button (a element)
Editor (textbox, textarea and label elements)
Footer (div element) with Sync button (a element)

Where Are We?

In this chapter we talked about the book's goals. We also established the overall design of the Notes application, defined its features, and created low fidelity user interface mock-ups that will help us build the app's main views and associated business logic.

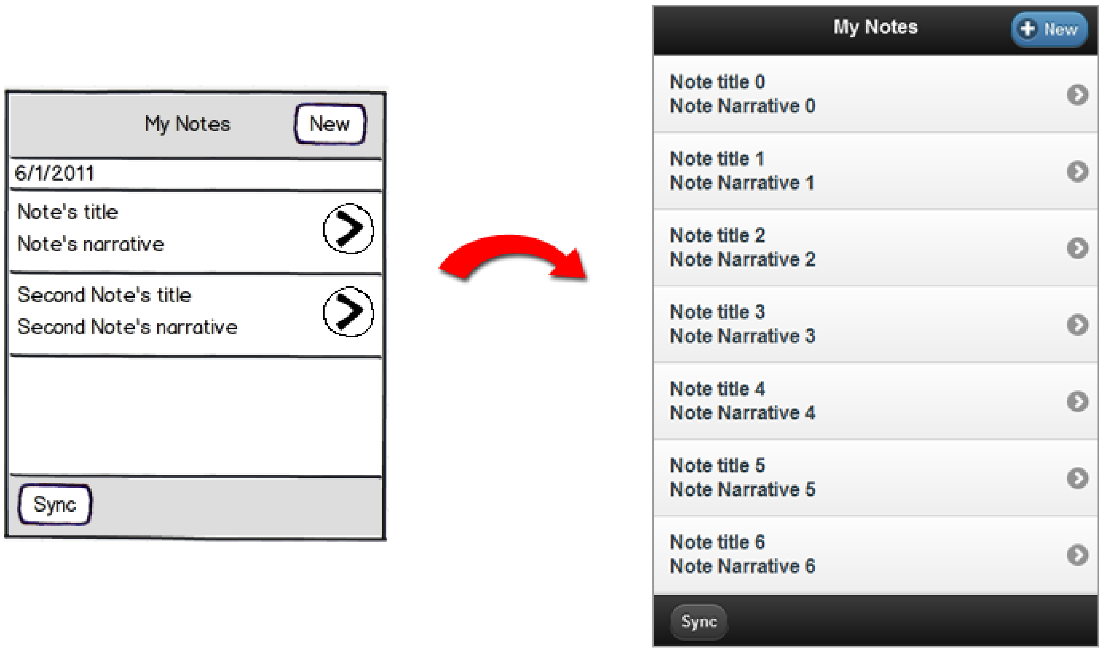
Let's now move on to building the Notes List View.

Chapter 2: Creating the Notes List

What You Will Learn In This Chapter

In this chapter, we are going to create the Notes List page, the page that renders the list of notes cached on the device. While building this page, we will become familiar with the following subjects:

- How jQuery Mobile works.
- How to build a jQuery Mobile page.
- How to execute code after a jQuery Mobile page loads.
- How to build a jQuery Mobile list view programmatically.
- How to define a Controller module that will feed data to jQuery Mobile pages, and coordinate their transitions.



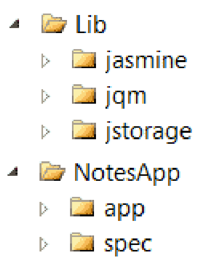
The Application's Directories and Files

Before we start writing code, let's spend a couple of minutes talking about our application's directories.

We will create a main directory for the application, which we will name *NotesApp*. This directory can be anywhere in your computer, as long as it is set up so it can be accessed from your local web server. Under *NotesApp* we will create an *app* directory, where we will place the business logic and presentation files; and a *spec* directory, where we will place our test suites.

At the same level of the *NotesApp* directory, we will create a *Lib* directory. *Lib* will contain directories for each of the libraries our application will use.

The directories should look as depicted below:



If you are curious about the *jasmine*, *jqm* and *jstorage* directories, these are the containers where we will place the Jasmine, jQuery Mobile and jStorage JavaScript frameworks. Don't worry about them for now. We will work on them later.

How jQuery Mobile Works

As its documentation clearly explains, jQuery Mobile is a unified user interface system with the following characteristics:

- It works seamlessly across all popular mobile device platforms.
- It uses jQuery and jQuery UI as its foundations.
- It has a lightweight codebase built on progressive enhancement.
- It has a flexible and easily themeable design.

A factor that differentiates jQuery Mobile from other frameworks is that it targets a wide variety of mobile browsers, 23 at the time of this writing. The reason this coverage is possible has to do with the way jQuery Mobile works.

jQuery Mobile works by applying CSS and JavaScript enhancements to HTML pages built with clean, semantic HTML. The usage of semantic HTML ensures compatibility with most web-enabled devices.

The techniques applied by the Framework to this HTML, transform the semantic page into a rich and interactive experience. We call these changes **progressive enhancements**, as they are applied progressively to the page, taking advantage of the capabilities of the browser on the web-enabled device.

The enhancements result in interactive pages, which provide a great user experience on the latest mobile browsers, and degrade gracefully on less capable browsers, without losing their intrinsic functionality.

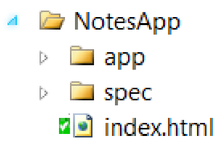
In addition, the Framework provides support for screen readers and other assistive technologies through a tight integration with the Web Accessibility Initiative - Accessible Rich Internet Applications Suite (WAI-ARIA) technical specification.

The Notes List jQuery Mobile Page

In order to build the Notes List, we need to create the HTML document that will host the Notes App.

jQuery Mobile allows us to build applications based on one or more HTML documents. The amount of markup we will write for our app is so small that we will not need multiple documents. The entire application's markup will reside in a single HTML file.

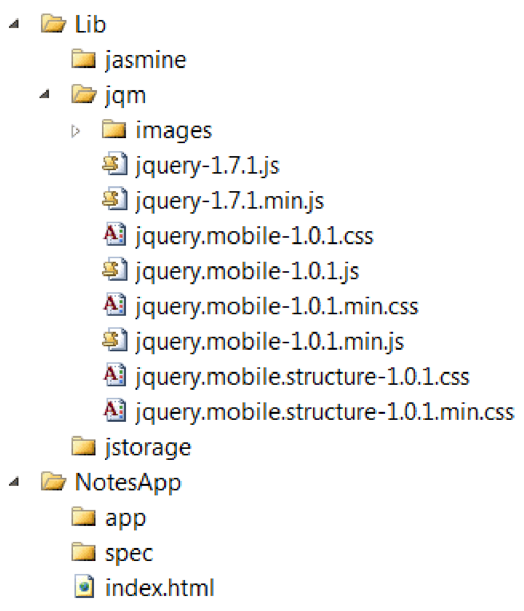
We will use jQuery Mobile's **page template** to create our app's HTML file. Let's create the index.html file in our application's directory, as depicted below:



We will add the following markup to the index.html file:

```
<html>
<head>
    <title></title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link href="../../lib/jqm/jquery.mobile-
1.3.0.min.css" rel="stylesheet" type="text/css" />
    <script src="../../lib/jqm/jquery-1.8.2.min.js" type="text/javascript"></script>

    <!--- Add controller module here --->

    <script src="../../Lib/jqm/jquery.mobile-
1.3.0.min.js" type="text/javascript"></script>
</head>
<body>
    <div data-role="page" id="notes-list-page" data-title="My Notes">
        <div data-role="header" data-position="fixed">
            <h1>
                My Notes</h1>
            <a href="#note-editor-page" class="ui-btn-right" data-theme="b" data-
icon="plus">New</a>
        </div>
        <div data-role="content" id="notes-list-content">
        </div>
         <div data-role="footer" data-position="fixed" class="ui-bar">
            <a id="sync-notes-button">Sync</a>
        </div>
    </div>
</body>
</html>
```

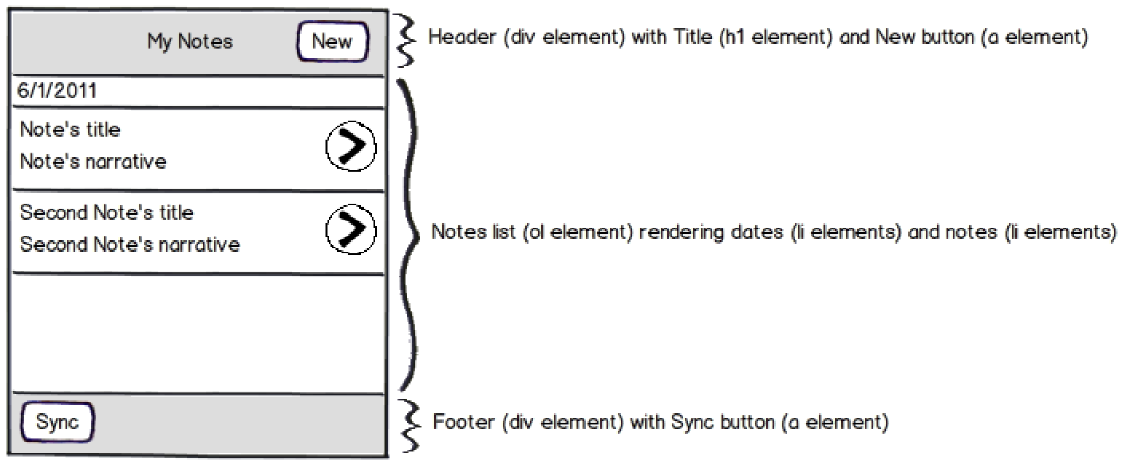
The *head* section of the file contains references to the jQuery and jQuery Mobile libraries. Now is a good time to download them from the jQuery and jQuery Mobile websites, and place them in the *jqm* directory:



```
◢ Lib
    jasmine
  ◢ jqm
    ▷ images
      jquery-1.7.1.js
      jquery-1.7.1.min.js
      jquery.mobile-1.0.1.css
      jquery.mobile-1.0.1.js
      jquery.mobile-1.0.1.min.css
      jquery.mobile-1.0.1.min.js
      jquery.mobile.structure-1.0.1.css
      jquery.mobile.structure-1.0.1.min.css
    jstorage
◢ NotesApp
    app
    spec
    index.html
```

In the *head* section of the index.html file we have also added a placeholder for the first JavaScript module we will work on, the *controller* module.

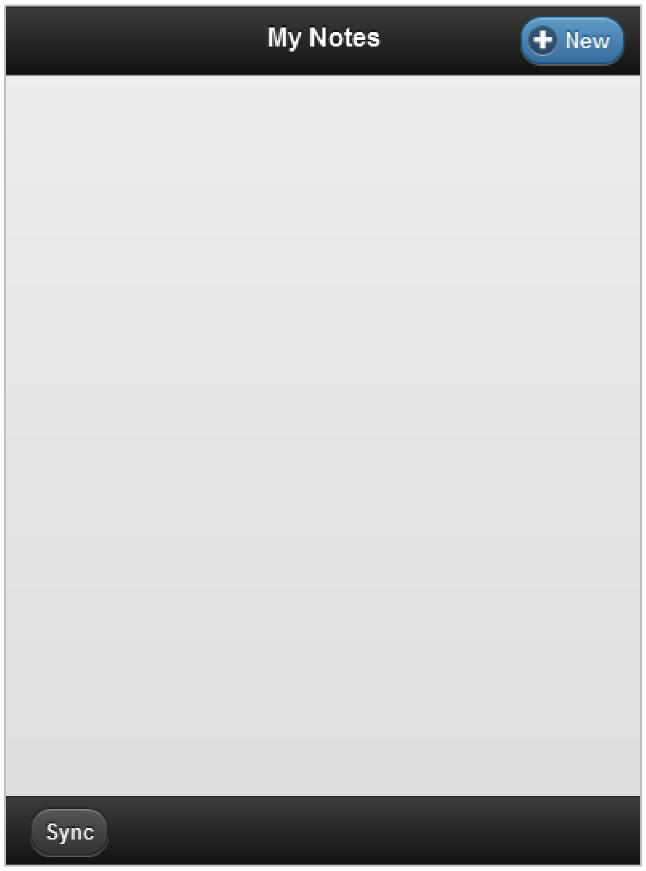
Let's shift our focus to the *body* section of the document, and take a minute to compare the code in this section to the page's mock-up, which we defined in Chapter 1:



My Notes
New
6/1/2011
Note's title
Note's narrative
Second Note's title
Second Note's narrative
Sync
Header (div element) with Title (h1 element) and New button (a element)
Notes list (ol element) rendering dates (li elements) and notes (li elements)
Footer (div element) with Sync button (a element)

They look similar, right?

As the markup indicates, we have created a single jQuery Mobile page with a header, content, and footer sections. This will be the Notes List page, the main page of the application.



A jQuery Mobile page and an HTML page are not the same thing. An HTML page is generally a physical document containing HTML markup. A jQuery Mobile page is a markup fragment that you can easily identify by the use of the *data-role="page"* attribute. An HTML document can contain one or more jQuery Mobile pages.

In the jQuery Mobile page that represents the Notes List, the header bar is defined with the *data-role="header"* attribute, the content area with *data-role="content"*, and the footer bar with *data-role="footer"*.

As the jQuery Mobile documentation explains, the Framework uses the attributes starting with the *data-* prefix to transform basic markup into and enhanced and styled widget.

The header bar of the Notes List page already contains the New button, which will allow our users to create a new note. The button is simply an anchor element that is automatically enhanced by the Framework. As we defined it immediately after the title of the header bar (the *h1* element), the button will render to the right of the title.

```
<a href="#note-editor-page" class="ui-btn-right" data-theme="b" data-
icon="plus">New</a>
```

We have decorated the button with the *data-theme* and *data-icon* attributes. We are using the *data-theme="b"* assignment to give the button a different color, making it the default button on the page. The value for the *data-icon* attribute maps to one of the built-in icons that ship with the Framework. You can find these icons in the *Lib/jqm/images* directory.

The value of the button's *href* attribute is the *id* of the Note Editor page. Later, we will bind to the *tap* event of the New button, and use the *href* to trigger a transition to the Note Editor page.

The content section of the Notes List page, a *div* adorned with the *data-role="content"* attribute, will host the list of cached notes:

```
<div data-role="content" id="notes-list-content">
</div>
```

In a few minutes, we will write code that inserts the cached notes list into this section.

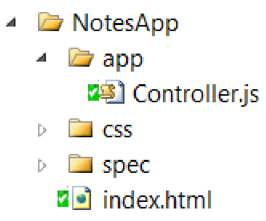
The footer bar contains the Sync button, an anchor element that will allow our users to send the updated notes to the server. We will bind to its *tap* event in order to trigger the notes synchronization.

This is the entire HTML required for the Notes List page. We can now start working on the JavaScript code that will render the notes cached on the device.

Defining the Controller Module

Our application will have a core that will control the behavior of the views. We will place this core in a JavaScript module, the *controller* module.

The application's only Controller (remember we're using the Model-View-Controller pattern) goes in a new file that we will name Controller.js. We will place this file in the *app* directory:



Back in the index.html file, we need to include the Controller's file:

```
<!--- Add controller module here --->
<script src="app/Controller.js" type="text/javascript"></script>
```

In the Controller.js file, let's define an empty *controller* module like so:

```
var Notes = Notes || {};

Notes.controller = (function () {


})();
```

We will also define an *init* function that will allow us to perform initialization code within the Controller.

```
var Notes = Notes || {};

Notes.controller = (function () {

    var init = function () {

    };

    var public = {
        init: init
    };

    return public;

})();
```

We want to trigger the controller's *init* function when jQuery Mobile starts to execute, this is why we will bind to the *mobileinit* jQuery Mobile event in the controller.js file, right after the *controller*'s definition:

```
var Notes = Notes || {};

Notes.controller = (function () {

    var init = function () {

    };

    var public = {
        init: init
    };

    return public;

})();

$(document).bind("mobileinit", function () {
    Notes.controller.init();
});
```

You would expect that the next step in the Controller's *init* function would be to render the cached notes. That is not wrong, but it would only work for rendering the notes upon application initialization. We're going to do something more useful. We're going to use the jQuery Mobile's *pagechange* event to render the cached notes, not only when the application starts, but also when the user transitions from the Note Editor page, which we haven't created yet, to the Notes List page.

How To Execute Code after A jQuery Mobile Page Loads

The *pagechange* event is triggered after the jQuery Mobile function *changePage* has finished loading the page into the DOM and all page transition animations have completed. This is the perfect moment for us to render the list of cached notes.

Let's modify the *controller* module, binding to jQuery Mobile's *pagechange* event like so:

```
var Notes = Notes || {};

Notes.controller = (function ($) {

    var notesListPageId = "notes-list-page";
    var notesListSelector = "#notes-list-content";

    var init = function () {

        var d = $(document);
        d.bind("pagechange", onPageChange);
    };

    var public = {
        init: init
    };

    return public;

})(jQuery);

$(document).bind("mobileinit", function () {
    Notes.controller.init();
});
```

The first change we have made in the *controller* module is the injection of the *jQuery* object at invocation time. This will allow us to have a *jQuery* reference correctly scoped within the module.

We have also added the *notesListPageId* and *notesListSelector* variables, which will allow us to reference the Notes List page and its content area.

In the *init* function, we acquire a reference to the *document* object and then bind to the *pagechange* event, defining an *onPageChange* function as the handler for the event:

```
var d = $(document);
d.bind("pagechange", onPageChange);
```

Now we can create *onPageChange*, right before *init*, as follows:

```
var onPageChange = function (event, data) {

    var toPageId = data.toPage.attr("id");

    switch (toPageId) {
        case notesListPageId:

            renderNotesList();
            break;
    }
};
```

This handler is relatively simple. It inspects the *id* attribute of the page we are transitioning to, available through the *data.toPage* property defined by jQuery Mobile, and takes action based on its value. If the *id* matches that of the Notes List page, the handler invokes a *renderNotesList* private function, which we will define next.

Creating a jQuery Mobile List Programmatically

The first version of *renderNotesList* function, which we will add before *onPageChange*, will simply render a few dummy notes to the Notes List page:

```
var renderNotesList = function () {

    var dummyNotesCount = 10,
        note,
        i;

    var view = $(notesListSelector);
    view.empty();

    var ul = $("<ul id=\"notes-list\" data-role=\"listview\"></ul>").appendTo(view);

    for (i = 0; i < dummyNotesCount; i += 1) {

        $("<li>"
        + "<a href=\"index.html#note-editor-page?noteId=" + i + "\">"
        + "<div>Note title " + i + "</div>"
        + "<div class=\"list-item-narrative\">Note Narrative " + i + "</div>"
        + "</a>"
        + "</li>").appendTo(ul);
    }

    ul.listview();

};
```

There isn't a lot going on in this function. We first use the *view* variable to store a reference to the area where we will render the notes, and then remove any existing HTML elements with a call to *empty*. Although not necessary when the application launches, we need this step in order to refresh the list after a note is added, deleted or updated.

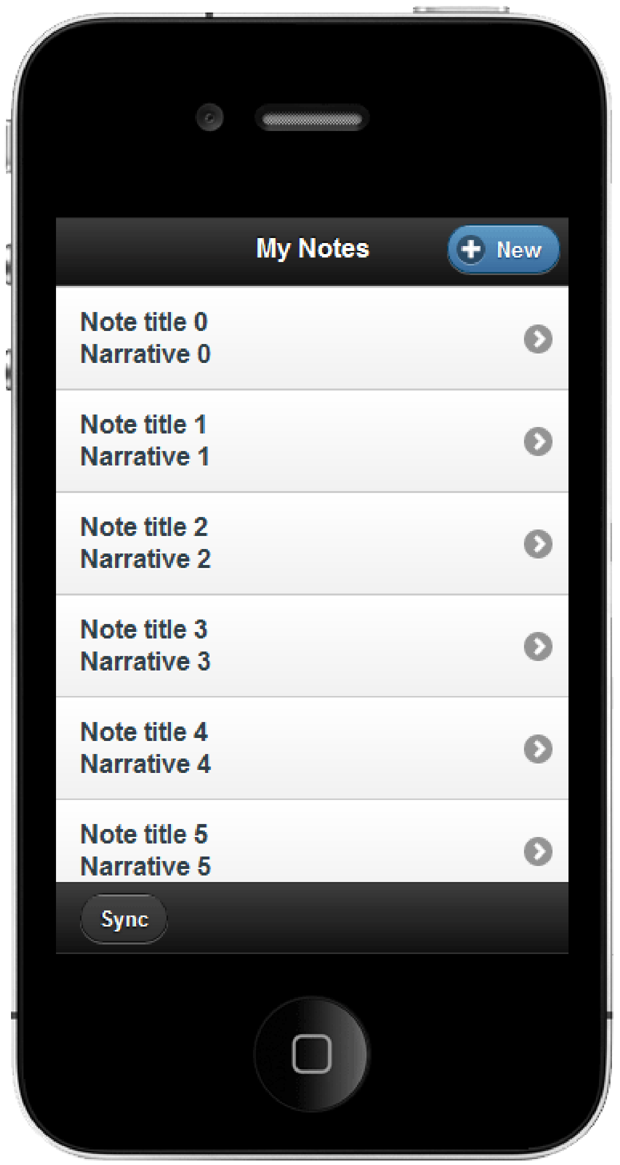
As you can see, we will use an unordered list to render the notes to the page. We are keeping a reference to the list in the *ul* variable, appending a *li* element to it for each note we need to render.

Each *li* element in turn contains a link to the Note Editor page, which specifies the *id* of the rendered note through the *noteId* parameter of the query string. Inside this link, we have defined a couple of *div* elements, which contain the note's title and narrative respectively. We have assigned each *div* a CSS class that will allow us to control its look. We will define these classes in a few minutes.

The last step, *ul.listview*, calls jQuery Mobile's *listview* plugin, which enhances the HTML list with the styles and behaviors defined for lists. This is what gives the list its mobile-friendly look and feel.

The most common way to generate a jQuery Mobile list view is through an unordered list, decorated with the *data-role="listview"* attribute. Many enhancements to HTML elements occur when jQuery Mobile loads, before the *document.ready* event fires. However, we are adding our list to the DOM long after the Framework is loaded, which means that we have to ask jQuery Mobile to enhance the list. This is the purpose of the *listview* call.

At this point, we are ready to check how our list looks. Let's fire our favorite WebKit browser or emulator, and review the page. The emulator should render a page similar to this:



Where Are We?

In this chapter, we learned how jQuery Mobile does its magic. We did so while building the Notes List page, which is the mobile page that will render the list of notes.

We also defined the *controller* module. This module will handle events generated in the application's views. Additionally, we learned how to create a jQuery Mobile list programmatically.

The Notes List page we built in this chapter renders a set of dummy notes. Now we are ready to feed real notes to this page, saved on the device, and retrieved by the next module we will build: the *dataContext* module.

Chapter 3: Retrieving and Rendering Cached Notes

What You Will Learn In This Chapter

We just built the Notes List page. Along with this page, we created the *controller* module, which is currently sending dummy notes to the page. It is time now to teach the app how to retrieve and render notes saved in HTML5 local storage.

Here's what we will learn in this chapter:

- How to build an application module to take care of data access operations.
- How to define and test the data access behavior of the application.
- How to retrieve data from HTML5's local storage.
- How to style list items in a jQuery Mobile list view.

Defining And Testing the Application's Business Logic

An important advantage of building the application in a modular fashion is that it helps us define and test the business logic independently of the presentation layer. To work on the business logic, we will follow these simple steps:

1. Define each application behavior using specifications.
2. Implement the behavior.
3. Test and confirm the behavior meets the specifications.

When it comes to testing, we could roll out our own testing Framework, or use one of the great frameworks that already exist. As this is not a book on how to build a Testing Framework, we will just pick the Jasmine Framework (https://github.com/pivotal/jasmine/wiki) and use it to test our app.

Jasmine is a Framework that allows us to test JavaScript code in a behavior-driven manner. This means that with Jasmine we can write tests in a natural language that describes the purpose and benefit of the code under test. This makes it easy for you as developer, and other stakeholders, to focus on the reason and purpose of the code rather than its technical details.

This example of a test will help you understand how Jasmine works:

```
describe("Notes functions", function () {
    it("Should return a NoteModel instance", function () {

        var note = Notes.app.createNote();

        expect(note instanceof Notes.model.NoteModel).toBeTruthy();
    });
});
```

We are using various Jasmine features in this example. One is the function *describe*, which we use to create a container – "Suite" in Jasmine parlance - for our specifications.

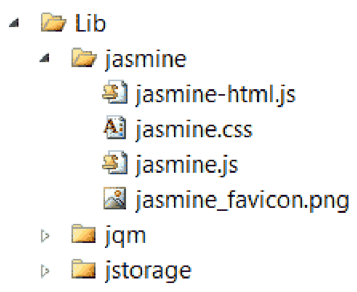
The concept of a specification, or spec, is implemented with the function *it*. When we call *it*, we pass a string describing the specification, as well as a function that defines a test for the spec. In the example, we are expressing that the code under test should be able to create and return a *NoteModel* instance.

The third feature we are using is the *expect* function, which defines an expectation about the behavior of the application. In order to express what result will make the test pass, we chain an expectation matcher to the *expect* call. In the example, we use the *toBeThruthy* matcher to express that our test will pass if the results are true.

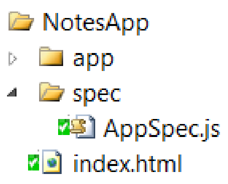
Within the *it* function, we will generally write the code needed to set up the test, as well as one or more calls to *expect*, which will define the expectations.

Getting Ready to Use Jasmine

Let's download Jasmine (http://pivotal.github.com/jasmine/), and place its files in the *Lib/jasmine* directory:

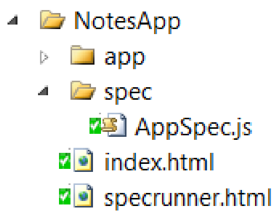


Now, we will create a source file for the Jasmine test suites. We will name this file AppSpec.js, and place it in the *spec* directory as depicted below:



Finally, let's create the specrunner.html. We will use this file to run our Jasmine test suites. The file is included in the Jasmine download, and we will need to modify it so it contains references to all the libraries used by the modules under test.

We will place specrunner.html in the *NotesApp* directory:



Here's its source:

```
<html>
<head>
    <title>Jasmine Test Runner</title>
    <!-- Libraries -->
    <script src="../lib/jqm/jquery- 1.7.1.min.js" type="text/javascript"></script>

    <!-- Jasmine -->
    <link href="../lib/jasmine/jasmine.css" rel="stylesheet" type="text/css" />
    <script src="../lib/jasmine/jasmine.js" type="text/javascript"></script>
    <script src="../lib/jasmine/jasmine-html.js" type="text/javascript"></script>
    <!-- App -->
    <!-- Modules under test -->
    <!-- Spec -->
    <script src="spec/AppSpec.js" type="text/javascript"></script>
</head>
<body>
    <script type="text/javascript">

        jasmine.getEnv().addReporter(new jasmine.TrivialReporter());

        jasmine.getEnv().execute();

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

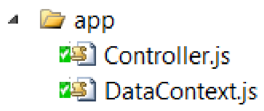
The file contains references to the Jasmine Framework, along with a placeholder for the modules we will test. We have also added an entry for the AppSpec.js specifications file.

Now we are ready to start building the business logic of the application.

The dataContext Module

The *dataContext* module will contain the business logic of the application. As we have discussed, to build this module, we will first define the specification for a given feature in the specifications file, then add the feature to the module, and last, confirm that feature meets the specification.

We will define the module in the DataContext.js file, which goes in the *app* directory:

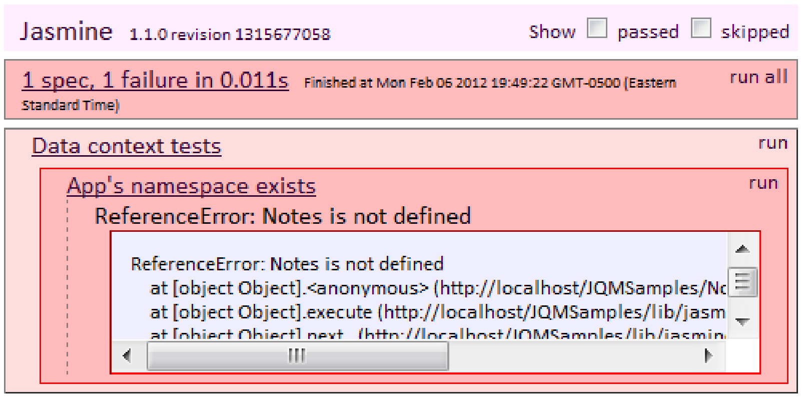


In order to use the *dataContext* module, the application's namespace must exist. This is the first specification we will create in the AppSpec.js file:

```
describe("Data context tests", function () {

    it("App's namespace exists", function () {
        expect(Notes).toBeDefined();
    });
});
```

The interesting part of this spec is how we use the *toBeDefined* expectation matcher to assert that the *Notes* namespace exists.

Let's open the specrunner.html file in our favorite browser and examine the output of the test. The results should look similar to these:



1 spec, 1 failure in 0.011s Finished at Mon Feb 06 2012 19:49:22 GMT-0500 (Eastern Standard Time)

Data context tests

App's namespace exists

ReferenceError: Notes is not defined

```
ReferenceError: Notes is not defined
    at [object Object].<anonymous> (http://localhost/JQMSamples/Nc
    at [object Object].execute (http://localhost/JQMSamples/lib/jasm
```

Our expectation of a defined application namespace just failed. This is correct, because we haven't defined the namespace yet.

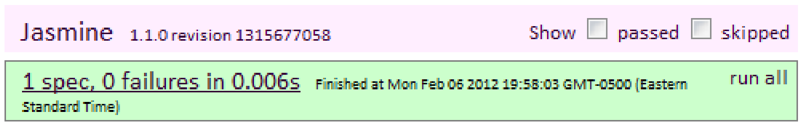
Let's define the *Notes* namespace in the DataContext.js file like so:

```
var Notes = Notes || {};
```

We also need to include DataContext.js in the head section of the specrunner.html file:

```
<!-- Modules under test -->
<script src="app/DataContext.js" type="text/javascript"></script>
```

If we run the test now, it should pass:



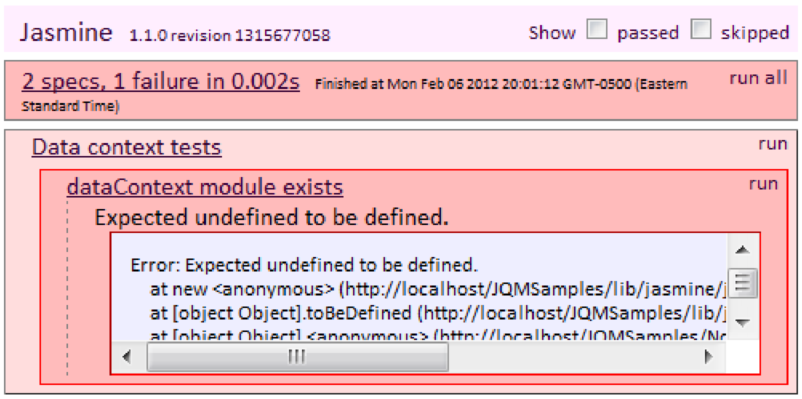
```
1 spec, 0 failures in 0.006s Finished at Mon Feb 06 2012 19:58:03 GMT-0500 (Eastern Standard Time)
```

This is a simple test, but good enough for you to see the rhythm that we want to establish here: define behavior, test, implement behavior, and test again.

We have a similar expectation with respect to the *dataContext* module, which we can express with a second *it* function in the AppSpec.js file:

```
it("dataContext module exists", function () {
    expect(Notes.dataContext).toBeDefined();
});
```

After reloading the specrunner.html file, we will see results similar to these:

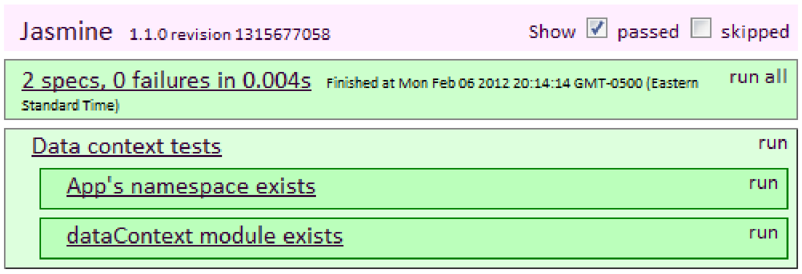


```
2 specs, 1 failure in 0.002s Finished at Mon Feb 06 2012 20:01:12 GMT-0500 (Eastern Standard Time)
Data context tests
  dataContext module exists
    Expected undefined to be defined.
      Error: Expected undefined to be defined.
        at new <anonymous> (http://localhost/JQMSamples/lib/jasmine/j
        at [object Object].toBeDefined (http://localhost/JQMSamples/lib/j
        at [object Object].<anonymous> (http://localhost/JQMSamples/Nc
```

Let's satisfy the expectation by defining an empty *dataContext* module in the DataContext.js file:

```
Notes.dataContext = (function () {
    return false;
})();
```

Now the test should pass:


Jasmine 1.1.0 revision 1315677058 Show ✓ passed ☐ skipped

2 specs, 0 failures in 0.004s Finished at Mon Feb 06 2012 20:14:14 GMT-0500 (Eastern Standard Time) run all

Data context tests run

App's namespace exists run

dataContext module exists run

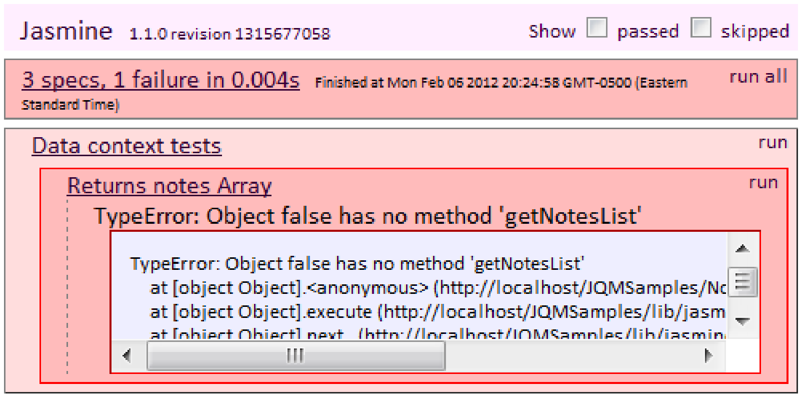
Moving on to more interesting behaviors, we want to define the ability to render the list of notes cached on the device. This means that the *dataContext* module must have the ability to pass a list of notes to the presentation layer.

We will define this behavior with a couple of Jasmine specifications. The first specification looks like this:

```
it("Returns notes Array", function () {

    var notesList = Notes.dataContext.getNotesList();

    expect(notesList instanceof Array).toBeTruthy();
});
```

Here we are defining the expectation that the *dataContext* module has a *getNotesList* function, which returns an instance of the *Array* class.

As we still have not defined this function, if we refresh specrunner.html in the browser, the test should fail:


Jasmine 1.1.0 revision 1315677058 Show ☐ passed ☐ skipped

3 specs, 1 failure in 0.004s Finished at Mon Feb 06 2012 20:24:58 GMT-0500 (Eastern Standard Time) run all

Data context tests run

Returns notes Array run

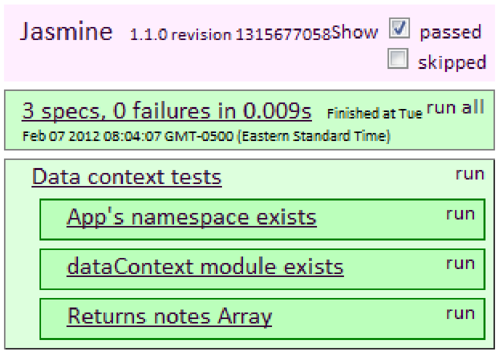
TypeError: Object false has no method 'getNotesList'

```
TypeError: Object false has no method 'getNotesList'
    at [object Object].<anonymous> (http://localhost/JQMSamples/Nc
    at [object Object].execute (http://localhost/JQMSamples/lib/jasm
    at [object Object].next_ (http://localhost/JQMSamples/lib/jasmin
```

We will fix this by adding the *getNotesList* function to the *dataContext* module:

```
Notes.dataContext = (function () {

    var notesList = [];

    var getNotesList = function () {
        return notesList;
    };

    var public = {
        getNotesList: getNotesList
    };

    return public;

})();
```

Refreshing specrunner.html should yield an all-green test:



```
3 specs, 0 failures in 0.009s  Finished at Tue Feb 07 2012 08:04:07 GMT-0500 (Eastern Standard Time)
Data context tests
    App's namespace exists
    dataContext module exists
    Returns notes Array
```

We now have a *dataContext* module with a *getNotesList* public function that returns an empty array. However, our ultimate goal is to retrieve the notes from the browser's local storage. To accomplish this we will need a class that represents a note.

The Note Model

Let's create the NoteModel.js file in the *app* folder, and add the following definition for the *NoteModel* class:

```
var Notes = Notes || {};

Notes.NoteModel = function(config) {
    this.id = config.id;
    this.dateCreated = config.dateCreated;
    this.title = config.title;
    this.narrative = config.narrative;
};
```

This construct should look familiar to you. The *NoteModel* is a class that represents a note. Every time we need to move a note's data around, we will use an instance of this class. What we will cache on the device is a serialized array of *NoteModel* instances.

We also need to include a reference to the NoteModel.js file in specrunner.html:

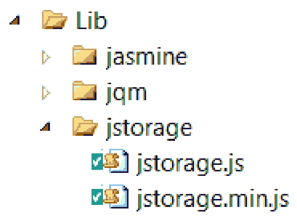
```
<!—Modules under test -->
<script src="app/DataContext.js" type="text/javascript"></script>
<script src="app/NoteModel.js" type="text/javascript"></script>
```

Retrieving Cached Notes from Local Storage

We are going to use HTML5's *localStorage* API to store notes across browser sessions. Local storage is based on keys and values, where the keys and values are strings. As we will cache an array of *NoteModel* instances, we will need a serialization mechanism to convert the array of notes to a string that will be saved in local storage. Similarly, we will need a mechanism to convert the serialized notes into an array of *NoteModel* instances when retrieving the cached notes.

We could write all the serialization and deserialization code we need. However, to simplify this tutorial, we will use an abstraction layer on top of the *localStorage* API. The layer that takes care of these services will be provided by the jStorage plugin (http://www.jstorage.info/).

The jStorage plugin allows us to cache numbers, strings and objects in local storage. We are going to place the jStorage plugin in the *Lib/jstorage* directory like so:



The specrunner.html file should reference the plugin like so:

```
<!-- Libraries -->
<script src="../../lib/jqm/jquery-1.8.2.min.js" type="text/javascript"></script>
<script src="../../lib/jstorage/jstorage.min.js" type="text/javascript"></script>
```

In the *dataContext* module, we will define the *loadNotesFromLocalStorage* private function, which will retrieve the cached notes from local storage:

```
var loadNotesFromLocalStorage = function () {

    var storedNotes = $.jStorage.get(notesListStorageKey);

    if (storedNotes !== null) {
        notesList = storedNotes;
    }
};
```

Let’s also declare the *notesListStorageKey* variable right after the *notesList* definition:

```
var notesList = [],
    notesListStorageKey;
```

The *loadNotesFromLocalStorage* function simply uses the jStorage plugin to retrieve the list of cached notes from local storage. The plugin takes care of converting the list of notes from a String instance to an Array instance, which is what we need.

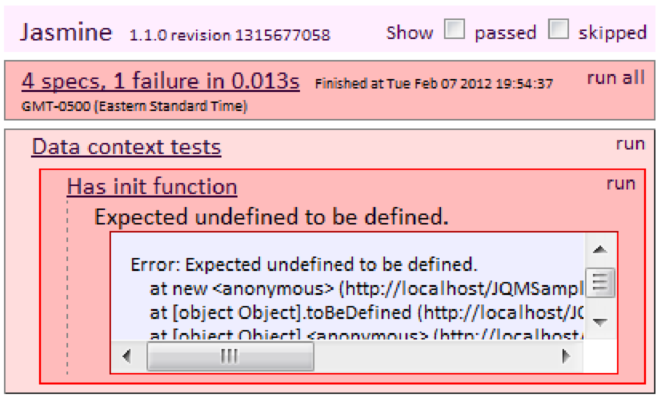
When the application runs, the cached notes will be immediately presented to the user. This means that the *loadNotesFromLocalStorage* is one of the first functions that will be called in the app.

To invoke this function when the application starts, as well as execute any other initialization code needed by the app, we will use a helper function, which we will name *init*.

Let’s work on the *init* function by first creating a spec for it in the AppSpec.js file:

```
it("Has init function", function () {
    expect(Notes.dataContext.init).toBeDefined();
});
```

As *init* is missing, the test should fail:



```
4 specs, 1 failure in 0.013s Finished at Tue Feb 07 2012 19:54:37 GMT-0500 (Eastern Standard Time)
Data context tests
Has init function
Expected undefined to be defined.
Error: Expected undefined to be defined.
    at new <anonymous> (http://localhost/JQMSampl
    at [object Object].toBeDefined (http://localhost/JC
```

We are going to add the *init* public function to the DataContext.js file:

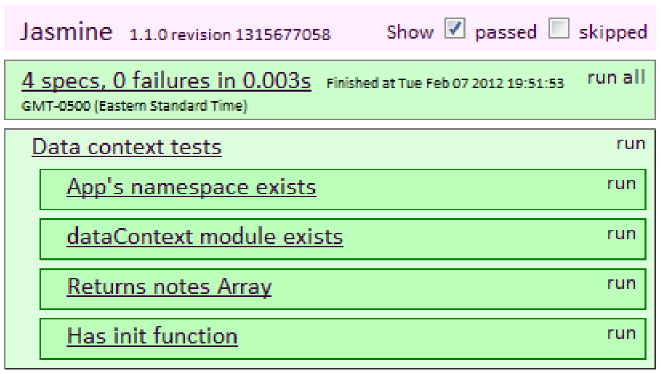
```
Notes.dataContext = (function () {

    var notesList = [],
        notesListStorageKey;

    var getNotesList = function () {
        return notesList;
    };

    var loadNotesFromLocalStorage = function () {

        var storedNotes = $.jStorage.get(notesListStorageKey);

        if (storedNotes !== null) {
            notesList = storedNotes;
        }
    };

    var init = function (storageKey) {
        notesListStorageKey = storageKey;
        loadNotesFromLocalStorage();
    };

    var public = {
        init: init,
        getNotesList: getNotesList
    };

    return public;

})();
```

Note how we are using *init* to initialize the *notesListStorageKey* variable:

```
var init = function (storageKey) {
    notesListStorageKey = storageKey;
    loadNotesFromLocalStorage();
};
```

Passing a value for the *notesListStorageKey* variable through the *init* function gives us the ability to use different local storage keys – one when *dataContext* is under test, and one when it is used by the *controller* module.

After this step, we can go back to the test, which this time should be green:


Jasmine 1.1.0 revision 1315677058

4 specs, 0 failures in 0.003s Finished at Tue Feb 07 2012 19:51:53 GMT-0500 (Eastern Standard Time)

Data context tests

App's namespace exists

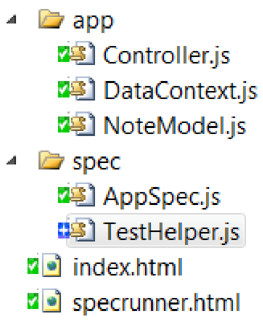
dataContext module exists

Returns notes Array

Has init function

Testing With Data by Retrieving a List of Dummy Notes

It would be nice if we could also test by saving a few dummy notes to local storage, and have the *dataContext* module retrieve them for us. This is not difficult to accomplish. Let's first create a *testHelper* module, which we will place in a new TestHelper.js file, in the *spec* folder:


```
app
    Controller.js
    DataContext.js
    NoteModel.js
spec
    AppSpec.js
    TestHelper.js
index.html
specrunner.html
```

The specrunner.html file needs to reference this module as well:

```
<!-- Modules under test -->
<script src="app/DataContext.js" type="text/javascript"></script>
<script src="app/NoteModel.js" type="text/javascript"></script>
<!-- Test Helper -->
<script src="spec/TestHelper.js" type="text/javascript"></script>
<!-- Spec -->
<script src="spec/AppSpec.js" type="text/javascript"></script>
```

The *testHelper* module will allow us to save a few dummy notes into local storage:

```
Notes.testHelper = (function () {

    var createDummyNotes = function (notesListStorageKey) {

        var notesCount = 10;
        var notes = [];

        for (var i = 0; i < notesCount; i++) {
            var config = {};
            var dateCreated = new Date();
            config.id = i.toString();
            config.title = "Title " + i;
            config.narrative = "Narrative " + i;
            config.dateCreated = dateCreated;


            var note = new Notes.NoteModel(config);

            notes.push(note);
        }

        $.jStorage.set(notesListStorageKey, notes);
    };

    var pub = {
        createDummyNotes: createDummyNotes
    };

    return pub;

})();
```

The notes created by the helper are saved to local storage through the jStorage plugin. Back in the AppSpec.js file, let's first define the storage key we will use for testing purposes. We will declare the key at the top of the module, before the specs are defined:

```
var notesListStorageKey = "Notes.NotesListTest";
```

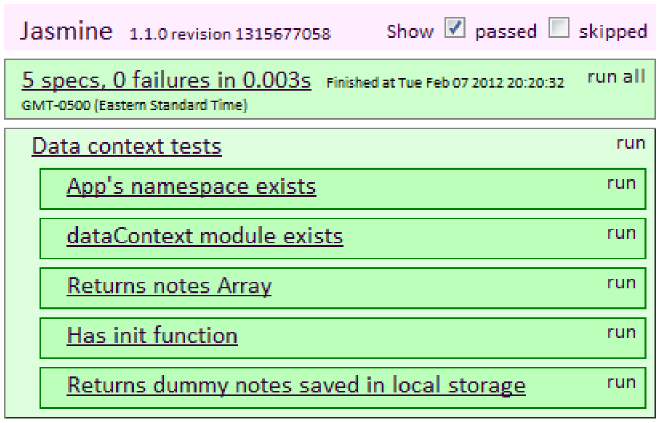
Then, we can define a spec that will confirm that we can load the dummy notes:

```
it("Returns dummy notes saved in local storage", function () {

    Notes.testHelper.createDummyNotes(notesListStorageKey);
    // Load dummy notes from localstorage.
    Notes.dataContext.init(notesListStorageKey);

    var notesList = Notes.dataContext.getNotesList();

    expect(notesList.length > 0).toBeTruthy();

    for (var i = 0; i < notesList.length; i+=1 ) {
        expect(notesList[i].id).toBeTruthy();
    }
});
```

This spec first uses the *testHelper*'s *createDummyNotes* function to place a few notes in local storage. Then, it invokes the data context's *init* function, which in turn calls the *loadNotesFromLocalStorage* private function. Next, the spec retrieves the notes list via a call to the *dataContext*'s *getNotesList* function. The expectation is simply that the notes list is not empty.

We could be more thorough and compare each of the retrieved notes to the ones we saved, making sure that the data did not change. I will leave this exercise as homework for you.

How about refreshing the specrunner.html file to check the test's output? We should see all green:



```
Jasmine 1.1.0 revision 1315677058                 Show ☑ passed ☐ skipped
5 specs, 0 failures in 0.003s  Finished at Tue Feb 07 2012 20:20:32   run all
GMT-0500 (Eastern Standard Time)
Data context tests                                                    run
  App's namespace exists                                              run
  dataContext module exists                                           run
  Returns notes Array                                                 run
  Has init function                                                   run
  Returns dummy notes saved in local storage                          run
```

Rendering Cached Notes

We have quickly moved from the *controller* module, where we fed the Notes List page a list of dummy notes, to the *dataContext* module, where we have added the ability to retrieve notes saved in the mobile browser's local storage.

It is time now to have the *controller* module request the list of notes from the *dataContext* module. The *controller* will then feed this list to the Notes List page.

How do we go about this? Well, the first step we need to take is to reference the jStorage plugin, as well as the NoteModel.js, DataContext.js and TestHelper.js files, in the index.html file:

```
<!-- Libraries -->
<script src="../../lib/jqm/jquery-1.8.2.min.js" type="text/javascript"></script>
<script src="../../lib/jstorage/jstorage.min.js" type="text/javascript"></script>
<!-- App -->
<script src="app/NoteModel.js" type="text/javascript"></script>
<script src="app/DataContext.js" type="text/javascript"></script>
<script src="app/Controller.js" type="text/javascript"></script>
<script src="spec/TestHelper.js" type="text/javascript"></script>
```

Next, we will modify the *mobileinit* event handler in the Controller.js file so it invokes the *testHelper* module's *createDummyNotes* function, which will give us a few test notes:

```
$(document).bind("mobileinit", function () {

    Notes.testHelper.createDummyNotes("Notes.NotesList");
    Notes.controller.init();
});
```

Note that we are using a different storage key here. Now we have two keys - one used for testing purposes, and one used by the application.

We also need to modify the *controller* so it is aware of the *dataContext* module. We will use a technique with which we are already familiar – dependency injection. Let's modify the *controller*'s definition so it takes the *dataContext* instance as an argument:

```
Notes.controller = (function ($, dataContext) {

    // Implementation omitted for brevity.

})(jQuery, Notes.dataContext);
```

Now we are injecting a reference to the *dataContext* into the *controller*. Let's also make sure that the *controller* calls the *dataContext*'s *init* function by inserting a line within the controller's own *init* function. We need this step because we want the *dataContext* module to load the notes saved in local storage when the controller is initialized:

```
var init = function () {

    dataContext.init(appStorageKey);
    var d = $(document);
    d.bind("pagechange", onPageChange);
};
```

We also need to declare the *appStorageKey* variable at the top of the *controller* module:

```
var appStorageKey = "Notes.NotesList";
```

What are we missing? Well, we need to revise the *renderNotesList* function in the controller module. This function should now use the *dataContext* module to retrieve the notes from local storage. Let's modify the function like so:

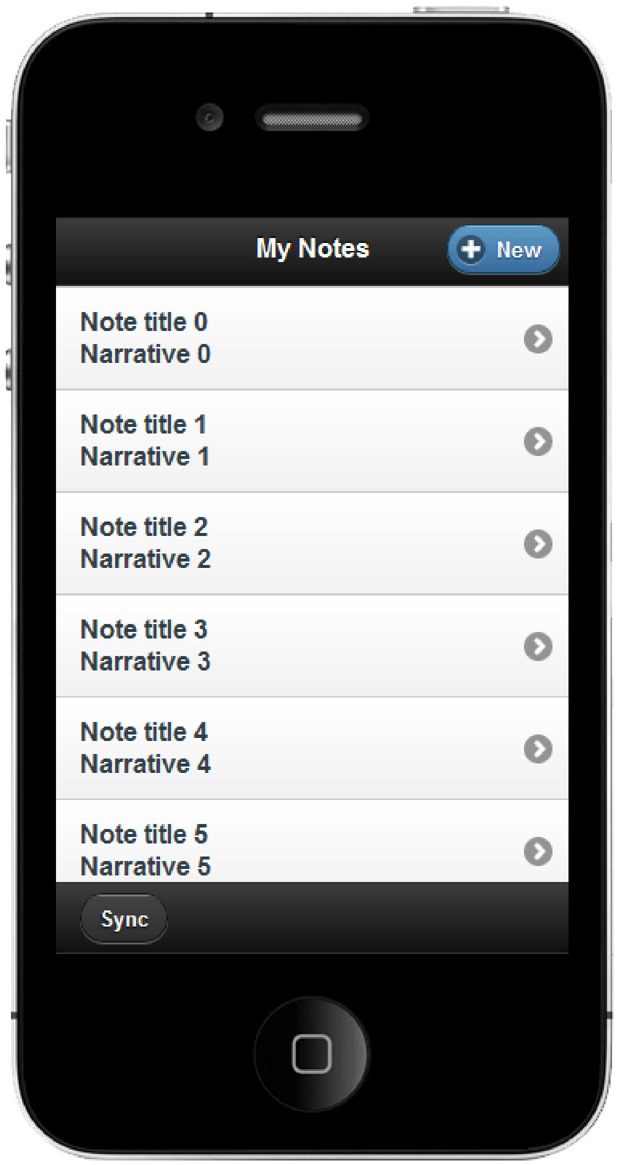
```
var renderNotesList = function () {

    var notesList = dataContext.getNotesList();
    var view = $(notesListSelector);

    view.empty();

    var liArray = [],
        notesCount,
        note,
        i,
        ul,
        liHtml;

    notesCount = notesList.length;
    ul = $("<ul id=\"notes-list\" data-role=\"listview\"></ul>").appendTo(view);

    for (i = 0; i < notesCount; i += 1) {

        note = notesList[i];

        liHtml = "<li>"
        + "<a data-url=\"index.html#note-editor-page?noteId=" + note.id + "\" 
href=\"index.html#note-editor-page?noteId=" + note.id + "\">"
        + "<div>" + note.title + "</div>"
        + "<div>" + note.narrative + "</div>"
        + "</a>"
        + "</li>"

        liArray.push(liHtml);

    }

    var listItems = liArray.join("");
    $(listItems).appendTo(ul);

    ul.listview();
};
```

The important change in this function is the addition of the line where we use the *dataContext* module to load the notes saved in local storage:

```
var notesList = dataContext.getNotesList();
```

After this step, we populate the jQuery Mobile list view by iterating through the array of notes.

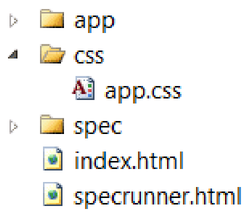
Let's inspect the results of our work. Open the index.html file in your favorite emulator or WebKit browser. The page should look like this:



We have accomplished an important project goal. At this point, the Notes List page is rendering notes retrieved from local storage. However, we still need to take care of two details that will greatly enhance the usability of this page. One is the formatting of the list items, and the other is grouping the notes by date.

How to Style List Items

Formatting the list items is a simple task. We will use a couple of css classes to define the look of the list items. Let's add the *css* directory to the application, and create an app.css file in it:



In the app.css file, we are going to define the *list-item-title* and *list-item-narrative* classes like so:

```
.list-item-title
{
	overflow: hidden;
	text-overflow: ellipsis;
}
.list-item-narrative
{
	color: #666666;
	font-weight: normal;
	overflow: hidden;
	text-overflow: ellipsis;
	min-height: 19px;
}
```

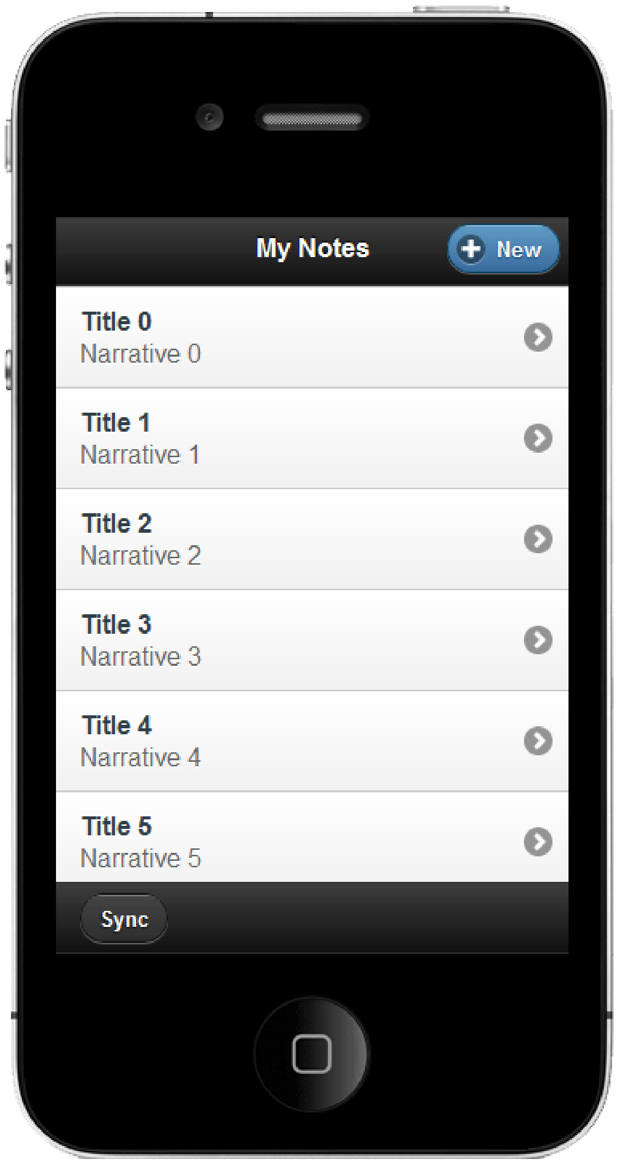
We also need to make sure the index.html file includes the app.css file:

```
<!-- Styles -->
<link href="css/app.css" rel="stylesheet" type="text/css" />
<!-- Libraries -->
<script src="../../lib/jqm/jquery-1.8.2.min.js" type="text/javascript"></script>
```

Now we need a small modification to the *renderNotesList* function in the *controller* module so it uses these styles when building the notes list:

```
var renderNotesList = function () {

    var notesList = dataContext.getNotesList();
    var view = $(notesListSelector);

    view.empty();

    var liArray = [],
        notesCount,
        note,
        i,
        ul,
        liHtml;

    notesCount = notesList.length;
    ul = $("<ul id=\"notes-list\" data-role=\"listview\"></ul>").appendTo(view);
    for (i = 0; i < notesCount; i += 1) {

        note = notesList[i];

        liHtml = "<li>"
        + "<a data-url=\"index.html#note-editor-page?noteId=" + note.id + "\"
href=\"index.html#note-editor-page?noteId=" + note.id + "\">"
        + "<div  class=\"list-item-title\">" + note.title + "</div>"
        + "<div class=\"list-item-narrative\">" + note.narrative + "</div>"
        + "</a>"
        + "</li>"

        liArray.push(liHtml);

    }

    var listItems = liArray.join("");
    $(listItems).appendTo(ul);


    ul.listview();
};
```

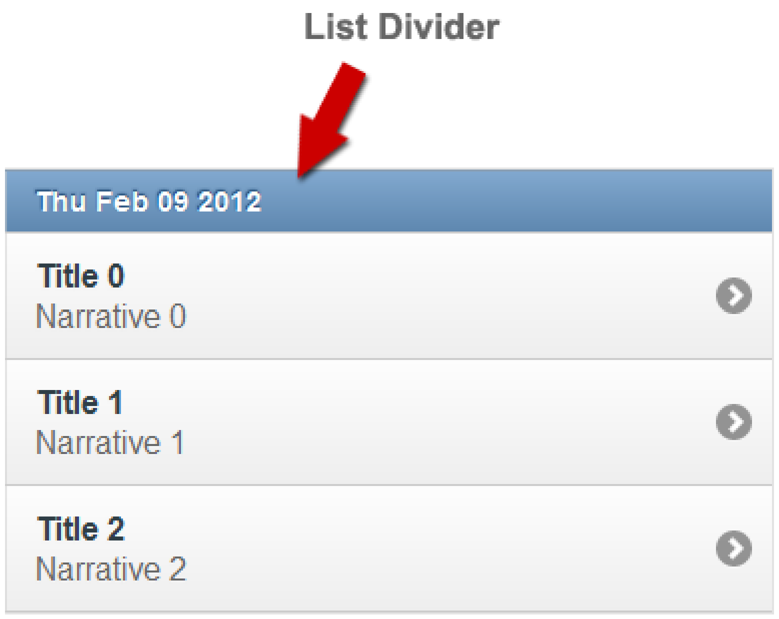
Notice how the *div* elements that contain the note's title and narrative are decorated with the styles we just created. Back in the emulator, the page should look like this:



Grouping Notes By Date

Grouping the notes by date is going to make it easier for the users of our app to find their notes. To accomplish this, we are going to use a jQuery Mobile feature that allows us to group list items – List dividers.

We can create a list divider by decorating a list item with the *data-role="list-divider"* attribute. This will change the style of the item, differentiating it from the rest of the items in the list.



The trick to adding the dividers to the list lies on inspecting the creation date of each note, and creating a divider for each new day. Let's revisit the *renderNotesList* function in the *controller* module, and add the code that creates the dividers:

```
var renderNotesList = function () {

    var notesList = dataContext.getNotesList();
    var view = $(notesListSelector);

    view.empty();

    var liArray = [],
        notesCount,
        note,
        i,
        ul,
        liHtml,
        dateGroup,
        noteDate;

    notesCount = notesList.length;
    ul = $("<ul id=\"notes-list\" data-role=\"listview\"></ul>").appendTo(view);

    for (i = 0; i < notesCount; i += 1) {

        note = notesList[i];

        noteDate = (new Date(note.dateCreated)).toDateString();

        if (dateGroup !== noteDate) {
            liArray.push("<li data-role=\"list-divider\">" + noteDate + "</li>");
            dateGroup = noteDate;
        }

        liHtml = "<li>"
        + "<a data-url=\"index.html#note-editor-page?noteId=" + note.id + "\"
href=\"index.html#note-editor-page?noteId=" + note.id + "\">"
        + "<div class=\"list-item-title\">" + note.title + "</div>"
        + "<div class=\"list-item-narrative\">" + note.narrative + "</div>"
        + "</a>"
        + "</li>"

        liArray.push(liHtml);

    }

    var listItems = liArray.join("");
    $(listItems).appendTo(ul);


    ul.listview();
};
```

We have introduced the *noteDate* and *dateGoup* variables. They will help us keep track of the days the notes were created. The *noteDate* value is the day the current note was created. The *dateGroup* value is the group to which the current note belongs.

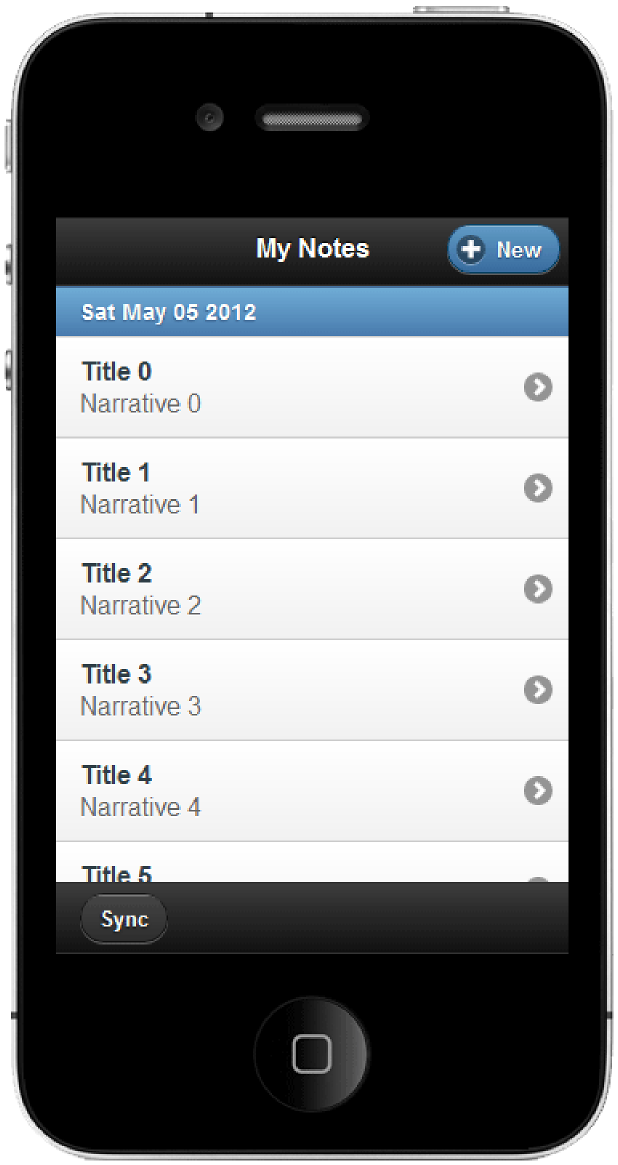
Inside the loop that creates the list items, we first grab the note's creation day:

```
noteDate = (new Date(note.dateCreated)).toDateString();
```

Then, we compare the day to the current group. If they are different, we add a list divider to the list, and update the current group with the value of the current note's creation day:

```
if (dateGroup !== noteDate) {
    liArray.push("<li data-role=\"list-divider\">" + noteDate + "</li>");
    dateGroup = noteDate;
}
```

This is all it takes to get the groups in place. Let's check it out on the emulator. Refreshing the page should produce a result similar to this:



Where Are We?

In this chapter, we learned how to retrieve data from HTML5's local storage. We used this knowledge to feed data to the Notes List page.

We also started building the *dataContext* module, which is in charge of the data access operations in the application. We are following a behavior-driven approach while building this module, first defining expectations for the behavior, and then adding features to the module in order to fulfill these expectations.

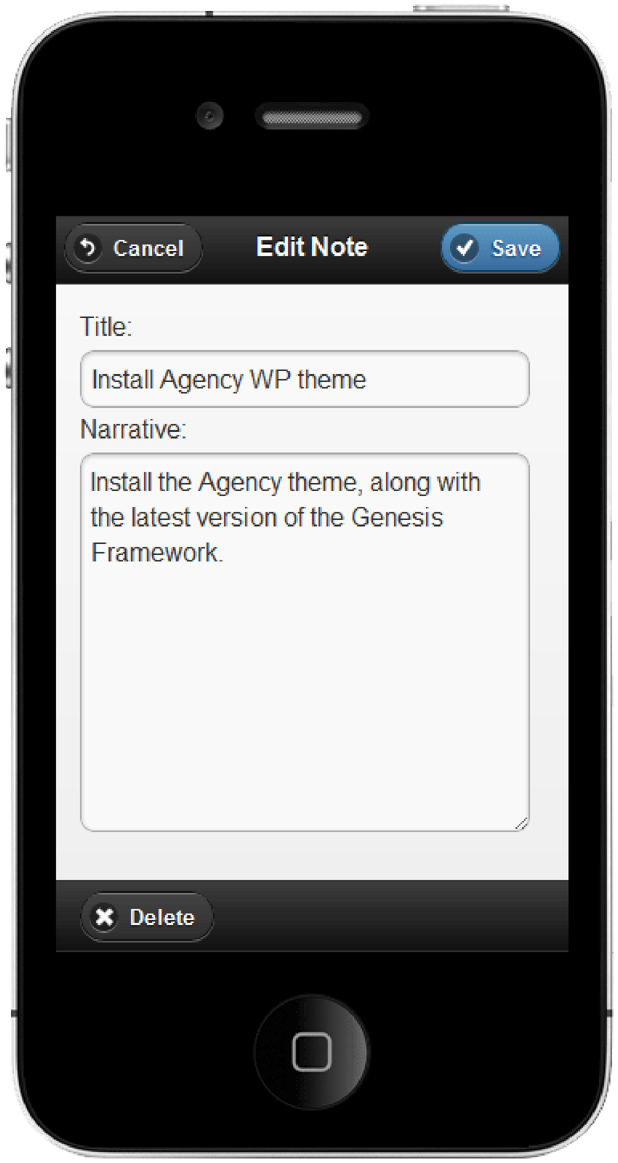
On the user interface side, we learned how to modify the look and feel of jQuery Mobile list views, changing their styles, and grouping items.

We have reached an important milestone. At this point, the application is able to present a list of cached notes upon launch. Our next steps will take us into the realm of form elements, as we are about to start working on the Note Editor page.

Chapter 4: Creating the Note Editor Page

What You Will Learn In This Chapter

The Note Editor page is very important in the application. This page will let our users create, edit and delete notes.



Title:

Install Agency WP theme

Narrative:

Install the Agency theme, along with the latest version of the Genesis Framework.

In this chapter, we are going to build the Note Editor page. While doing so, we will cover the following subjects:

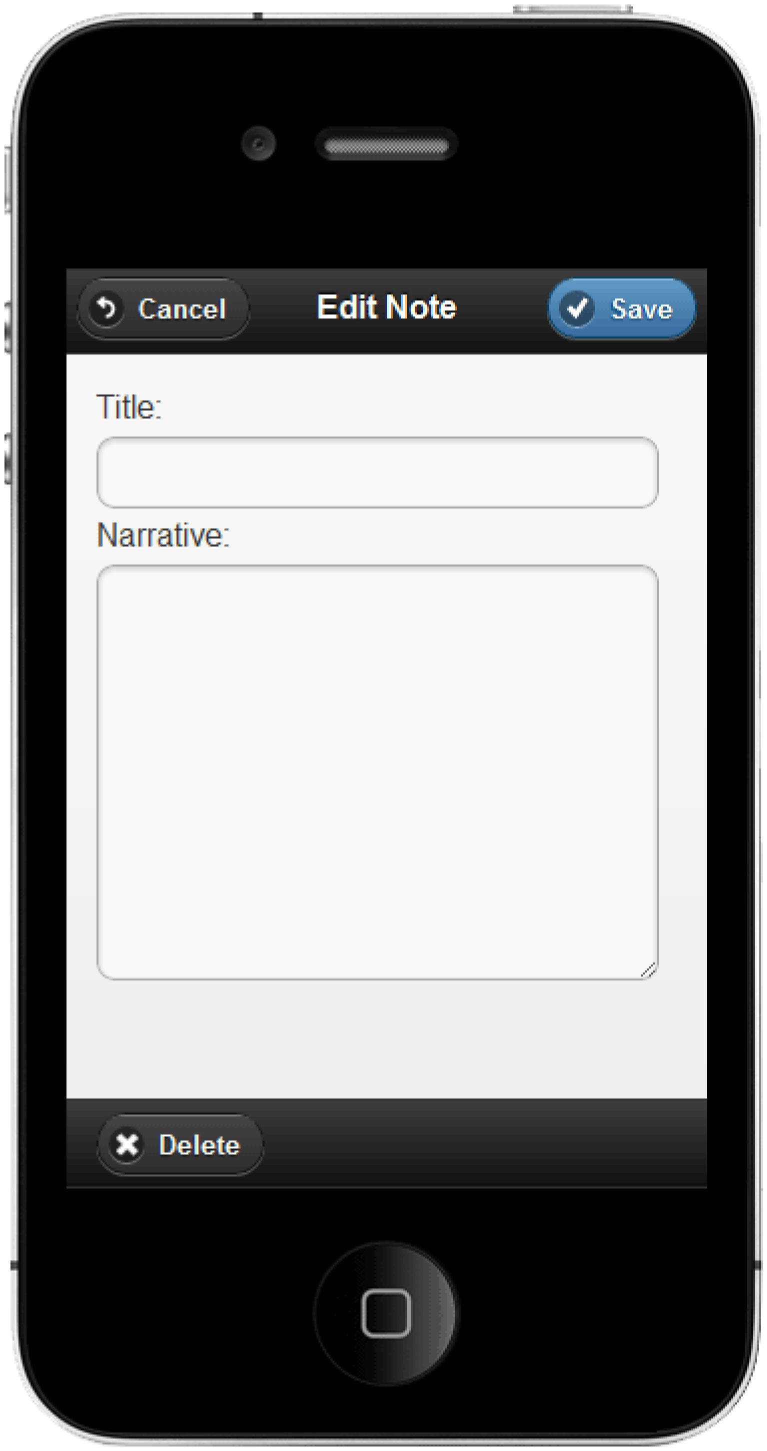
- How to create a jQuery Mobile form.
- How to pass data from a jQuery Mobile list view to a form.
- How to load data into the form.
- How to save data to HTML5's local storage.
- How to create jQuery Mobile dialogs.
- How to create a custom jQuery Mobile theme swatch.

Creating the Note Editor Page

Similar to the Notes List Page, The Note Editor Page is a *div* element with the *data-role="page"* attribute. We will add this *div* to the index.html file like so:

```
<!--Note Editor page-->
<div data-role="page" id="note-editor-page" data-title="Edit Note">
    <div data-role="header" data-position="fixed">
        <a href="#notes-list-page" data-icon="back" data-rel="back">Cancel</a>
        <h1>
            Edit Note</h1>
        <a id="save-note-button" href="" data-theme="b" data-icon="check">Save</a>
    </div>
    <div data-role="content">
        <form action="" method="post" id="note-editor-form">
        <label for="note-title-editor">
            Title:</label>
        <input type="text" name="note-title-editor" id="note-title-editor" value="" />
        <label for="note-narrative-editor">
            Narrative:</label>
        <textarea name="note-narrative-editor" id="note-narrative-editor"></textarea>
        </form>
    </div>
    <div data-role="footer" data-position="fixed" class="ui-bar">
        <a id="delete-note-button" data-icon="delete" data-transition="slideup" data-
rel="dialog">Delete</a>
    </div>
</div>
```

This page consists of a header area with the Save and Cancel buttons, a form with the elements that allow us to edit a note, and a footer that hosts the Delete button.



As you can see, jQuery Mobile has taken care of nicely styling and laying out the form elements.

We will give the Save and Delete button special treatment, binding to their *tap* events in order to trigger the routines that save or delete the note loaded in the Note Editor page.

The role of the Cancel button is to take us back to the Notes List page. The *data-rel=back* attribute causes taps on this button to go back one history entry and ignore the button's *href*. However, we still need the *href* to point to the *#notes-list-page* bookmark so the button still works in browsers where jQuery Mobile provides a basic, non-enhanced HTML experience.

Loading a Note in the Editor

Let us take a moment to think about what needs to happen in order to load a note into the editor. For existing notes, when the user taps the *li* element representing a note in the Notes List page, we will perform the following steps:

1. Lookup the note in the *notesList* array, based on the note's *id* contained in the *li* element's link.
2. Set the values of the *title* and *narrative* form elements in the Note Editor page to those of the selected note's *title* and *narrative*.
3. Make the Note Editor page active.

For new notes, when the user taps the New button in the Notes List page, we will perform these steps:

1. Reset the values of the *title* and *narrative* form elements in the Note Editor page.
2. Make the Note Editor page active.

To load a note, the first thing we need in our *controller* module is a reference to the Note Editor. Let's go back to the top of the *controller* module and define a selector for the Note Editor page:

```
var noteEditorPageId = "note-editor-page";
```

Next, we need to return to the *onPageChange* function and handle the case when we are switching to the editor page:

```
var onPageChange = function (event, data) {

    var toPageId = data.toPage.attr("id");
    var fromPageId = null;

    if (data.options.fromPage) {
        fromPageId = data.options.fromPage.attr("id");
    }

    switch (toPageId) {

        case notesListPageId:
            renderNotesList();
            break;

        case noteEditorPageId:

            if (fromPageId === notesListPageId) {
                renderSelectedNote(data);
            }
            break;
    }
};
```

The handler is a little bit more complex now. We added the *fromPageId* variable, which will store the *id* of the page from which we're navigating. We will call this page *source* page from now on. The page we're navigating to will be the *target* page.

The value of *fromPageId* will help us determine if we need to load a note into the editor:

```
if (fromPageId === notesListPageId) {
    renderSelectedNote(data);
}
```

If the source page is the Notes List, we load the note by calling the *renderSelectedNote* function, which we will implement in a minute.

Note that we added the check for the *fromPage* parameter because the *pagechange* event is also triggered after the application is launched, when there isn't a source page yet. It doesn't make sense to acquire the *id* of the source page when the source page itself doesn't exist:

```
if (data.options.fromPage) {
    fromPageId = data.options.fromPage.attr("id");
}
```

Before implementing *renderSelectedNote*, let's jump to the top of the *controller* module and create references to the *title* and *narrative* form elements:

```
var noteTitleEditorSel = "[name=note-title-editor]",
    noteNarrativeEditorSel = "[name=note-narrative-editor]";
```

The *renderSelectedNote* will take care of loading the selected note into the editor. Here's the implementation of the function:

```
var renderSelectedNote = function (data) {

    var u = $.mobile.path.parseUrl(data.options.fromPage.context.URL);
    var re = "^#" + noteEditorPageId;

    if (u.hash.search(re) !== -1) {

        var queryStringObj = queryStringToObject(data.options.queryString);

        var titleEditor = $(noteTitleEditorSel);
        var narrativeEditor = $(noteNarrativeEditorSel);

        var noteId = queryStringObj["noteId"];

        if (typeof noteId !== "undefined") {

            // We're editing an existing note. Load the note’s properties in the
editor.

        } else {
            // We're creating a note. Reset the fields.

        }

        titleEditor.focus();
    }
};
```

The *renderSelectedNote* function takes advantage of the fact that, in the *renderNotesList* function, we’re passing the selected note’s *id* in the query string of the list item’s link:

```
liArray.push("<li>"
                    + "<a data-url=\"index.html#note-editor-page?noteId=" + note.id +
"\" href=\"index.html#note-editor-page?noteId=" + note.id + "\">"
                    + "<div  class=\"list-item-title\">" + note.title + "</div>"
                    + "<div class=\"list-item-narrative\">" + note.narrative +
"</div>"
                    + "</a>"
                    + "</li>");
```

Our first goal inside *renderSelectedNote* is to inspect the *hash* of the source page’s URL, to make sure the source page is the Notes List page. We do this using a regular expression search:

```
var u = $.mobile.path.parseUrl(data.options.fromPage.context.URL);
var re = "^#" + noteEditorPageId;

if (u.hash.search(re) !== -1) {

    // Code omitted for brevity...
}
```

To acquire the URL’s hash, we use the *$.mobile.path.parseUrl* function, which parses a URL into an object that facilitates accessing the URL’s components. Once we’re certain the source page is the Notes List, we proceed to create an object containing the query string parameters passed from the source page. We use the *queryStringToObject* helper function for this task. Let's add it to the *controller* module:

```
var queryStringToObject = function (queryString) {

    var queryStringObj = {};
    var e;
    var a = /\+/g;  // Replace + symbol with a space
    var r = /([^&;=]+)=?([^&;]*)/g;
    var d = function (s) { return decodeURIComponent(s.replace(a, " ")); };

    e = r.exec(queryString);
    while (e) {
        queryStringObj[d(e[1])] = d(e[2]);
        e = r.exec(queryString);

    }

    return queryStringObj;
};
```

The *queryStringToObject* function takes the value of the *data.options.queryString* property as a parameter:

```
var queryStringObj = queryStringToObject(data.options.queryString);
```

The *data.options* object does not have a native *queryString* property. However, we can create it if we find a point in time, before the page transition occurs, where we can acquire the value of the query string.

It turns out that this is possible if we define a handler for jQuery Mobile's *pagebeforechange* event. Let's return to our *init* function and add the following line:

```
d.bind("pagebeforechange", onPageBeforeChange);
```

Now we can define *onPageBeforeChange* like so:

```
var onPageBeforeChange = function (event, data) {

    if (typeof data.toPage === "string") {

        var url = $.mobile.path.parseUrl(data.toPage);

        if ($.mobile.path.isEmbeddedPage(url)) {

            data.options.queryString = $.mobile.path.parseUrl(url.hash.replace(/^#/,
"")).search.replace("?", "");
        }
    }
};
```

Pay attention to the following line:

```
data.options.queryString = $.mobile.path.parseUrl(url.hash.replace(/^#/,
"")).search.replace("?", "");
```

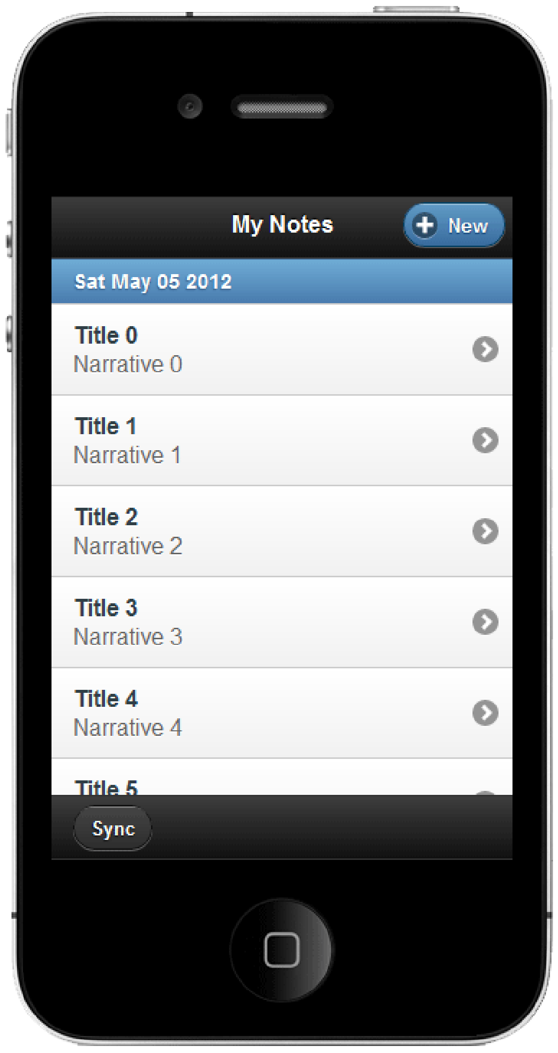
Here we use *$.mobile.path.parseUrl* to acquire the query string and add it to the *data.options* object. As the *onPageBeforeChange* handler is invoked before the *onPageChange* handler, we've found the right time to inject the query string defined in the source page into the events chain, and propagate it to the target pages where it can be used.

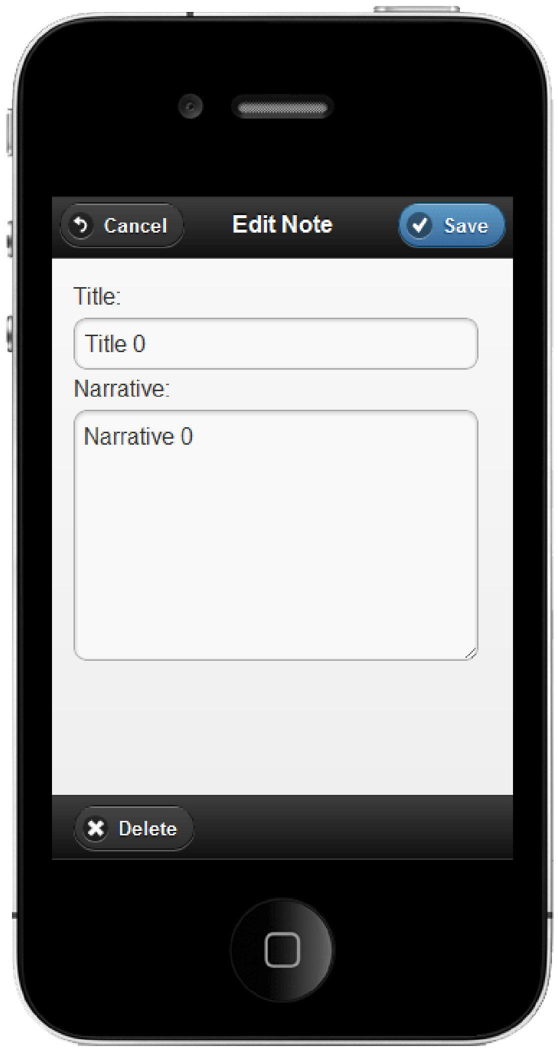
Back in *renderSelectedNote*, we can finally take care of loading the selected note, or resetting the *title* and *narrative* fields, like so:

```
var renderSelectedNote = function (data) {

    var u = $.mobile.path.parseUrl(data.options.fromPage.context.URL);
    var re = "^#" + noteEditorPageId;

    if (u.hash.search(re) !== -1) {

        var queryStringObj = queryStringToObject(data.options.queryString);

        var titleEditor = $(noteTitleEditorSel);
        var narrativeEditor = $(noteNarrativeEditorSel);

        var noteId = queryStringObj["noteId"];

        if (typeof noteId !== "undefined") {

            // We're editing an existing note. Load the note's properties in the
editor.
            var notesList = dataContext.getNotesList();
            var notesCount = notesList.length;
            var note;

            for (var i = 0; i < notesCount; i++) {

                note = notesList[i];

                if (noteId === note.id) {
                    currentNote = note;
                    titleEditor.val(currentNote.title);
                    narrativeEditor.val(currentNote.narrative);
                }
            }
        } else {
            // We're creating a note. Reset the fields.
            titleEditor.val("");
            narrativeEditor.val("");
        }

        titleEditor.focus();
    }
};
```

Observe how we're keeping a reference to the selected note in the *currentNote* variable. This will later allow us to save or delete the note without having to perform a lookup on the *notesList* array.

That is what it takes to load a note. Let's make sure things are working as expected. Fire up your favorite WebKit browser or emulator, and confirm that tapping a note in the Notes List page loads the note into the Note Editor page:





Similarly, tapping the New button should reset the editor's fields.

Saving a Note

The Save Note workflow is initiated when a user taps the Save button. The *controller* module needs to define a handler for the button's *tap* event. We will use this handler to invoke a *saveNote* function that we will create in the *dataContext* module. Let's work on the *tap* handler first.

We need a reference to the Save button, which we can create at the top of the *controller* module like so:

```
var saveNoteButtonSel = "#save-note-button";
```

Next, we need to define a *tap* handler for the button in the *controller*’s *init* function:

```
var init = function () {

    dataContext.init("Notes.NotesList");

    var d = $(document);
    d.bind("pagebeforechange", onPageBeforeChange);
    d.bind("pagechange", onPageChange);
    d.delegate(saveNoteButtonSel, "tap", onSaveNoteButtonTapped);
};
```

Now we can implement *onSaveButtonTapped* like so:

```
var onSaveNoteButtonTapped = function () {

    // Validate note.
    var titleEditor = $(noteTitleEditorSel);
    var narrativeEditor = $(noteNarrativeEditorSel);
    var tempNote = dataContext.createBlankNote();


    tempNote.title = titleEditor.val();
    tempNote.narrative = narrativeEditor.val();

    if (tempNote.isValid()) {

        if (null !== currentNote) {

            currentNote.title = tempNote.title;
            currentNote.narrative = tempNote.narrative;
        } else {

            currentNote = tempNote;
        }

        dataContext.saveNote(currentNote);

        returnToNotesListPage();

    } else {
        // TODO: Inform the user the note is invalid.
    }
};
```

The first interesting thing that happens in this function is the creation of a temporary note, invoking a *dataContext* function:

```
var tempNote = dataContext.createBlankNote();
```

The *createBlankNote* function does not exist in the *dataContext* module yet. Let's go ahead and define a test for it in the AppSpec.js file:

```
it("Returns a blank note", function () {

    var blankNote = Notes.dataContext.createBlankNote();
    expect(blankNote.title.length === 0).toBeTruthy();
    expect(blankNote.narrative.length === 0).toBeTruthy();
});
```

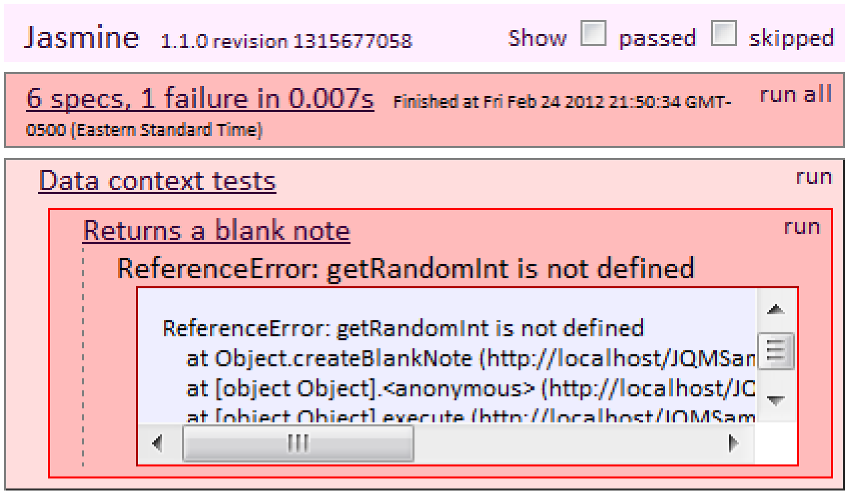
The test will not pass, as we need to add *createBlankNote* to the *dataContext*:

```
var createBlankNote = function () {

    var dateCreated = new Date();
    var id = dateCreated.getTime().toString() + (getRandomInt(0, 100)).toString();
    var noteModel = new Notes.NoteModel({
        id: id,
        dateCreated: dateCreated,
        title: "",
        narrative: ""
    });

    return noteModel;
};
```

We also need to add the function to the module's public interface:

```
var public = {
    init: init,
    getNotesList: getNotesList,
    createBlankNote: createBlankNote
};
```

If you examine *createBlankNote*, you will notice that it calls a *getRandomInt* helper function in order to create the id of the new note. As we haven't defined *getRandomInt* yet, if we run the Jasmine spec, the results should indicate that the helper function is missing:


Jasmine 1.1.0 revision 1315677058 Show ☐ passed ☐ skipped

6 specs, 1 failure in 0.007s Finished at Fri Feb 24 2012 21:50:34 GMT-0500 (Eastern Standard Time) run all

Data context tests run

Returns a blank note run

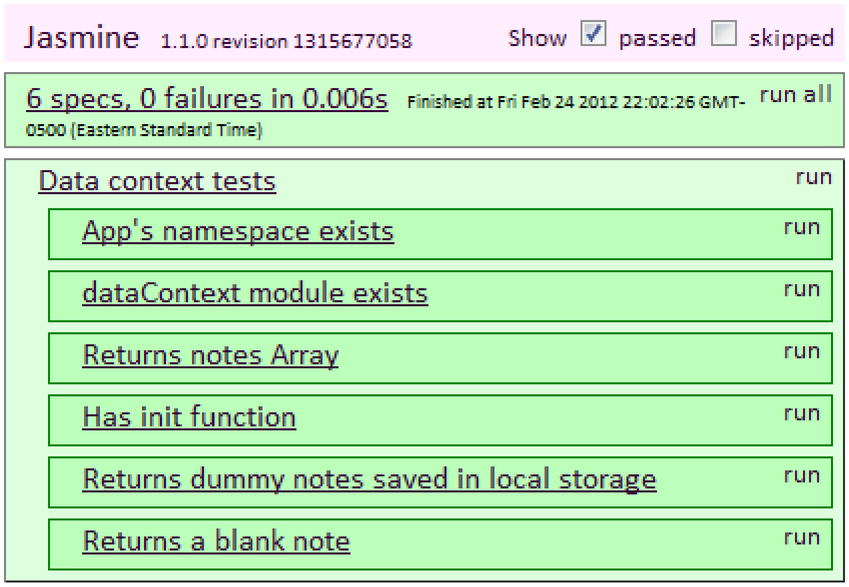
ReferenceError: getRandomInt is not defined

```
ReferenceError: getRandomInt is not defined
    at Object.createBlankNote (http://localhost/JQMSan
    at [object Object].<anonymous> (http://localhost/JC
    at [object Object].execute (http://localhost/JQMSam
```

Let's add *getRandomInt* to the *dataContext* module like so:

```
var getRandomInt = function(min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
};
```

After this modification, the test should pass:


Jasmine 1.1.0 revision 1315677058 Show ☑ passed ☐ skipped

6 specs, 0 failures in 0.006s Finished at Fri Feb 24 2012 22:02:26 GMT-0500 (Eastern Standard Time) run all

Data context tests run

App's namespace exists run

dataContext module exists run

Returns notes Array run

Has init function run

Returns dummy notes saved in local storage run

Returns a blank note run

Back in the controller's *onSaveNoteButtonTapped* handler, the second interesting thing is the call to the *NoteModel*'s *isValid* function:

```
if (tempNote.isValid()) {

    if (null !== currentNote) {
        currentNote.title = tempNote.title;
        currentNote.narrative = tempNote.narrative;
    } else {

        currentNote = tempNote;
    }

    dataContext.saveNote(currentNote);

    returnToNotesListPage();

} else {
    // TODO: Inform the user the note is invalid.
}
```

This function will allow us to validate a note before committing it to the cache. We are going to implement *isValid* in the NoteModel.js file. Here's the code:

```
Notes.NoteModel.prototype.isValid = function () {
    "use strict";
    if (this.title && this.title.length > 0) {
        return true;
    }
    return false;
};
```

A check on the *title* property is enough to validate the note. Nothing complicated.

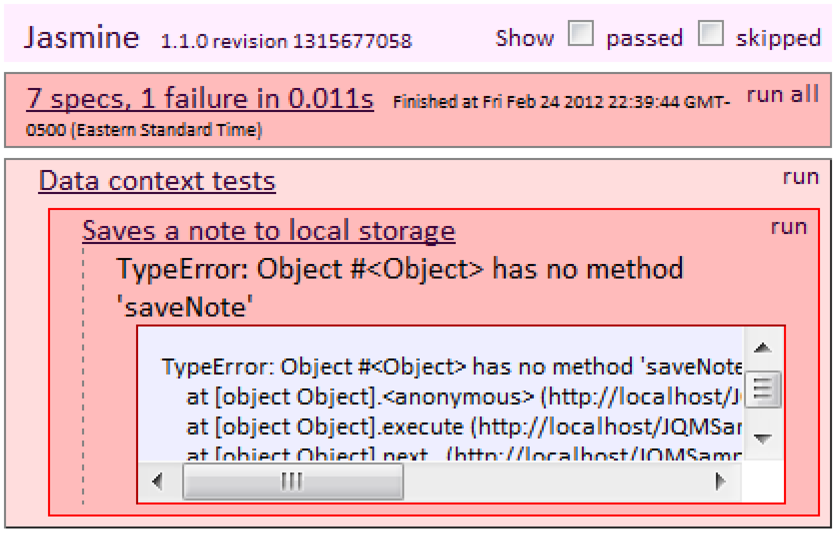
Back in *onSaveButtonTapped*, we find out if we're editing an existing note by observing the value of *currentNote*. If *currentNote* points to an existing note, we transfer the *title* and *narrative* of the temporary note to it. If *currentNote* is not pointing to an existing note, we make it point to the temporary note's reference. Then, we save the note, new or edited, by calling the *dataContext* module's *saveNote* function.

We haven't created *saveNote* yet, so, let's go ahead and define a behavior test for it in the AppSpec.js file:

```
it("Saves a note to local storage", function () {

    // Make sure LS is empty before the test.
    $.jStorage.deleteKey(notesListStorageKey);
    var notesList = $.jStorage.get(notesListStorageKey);
    expect(notesList).toBeNull();

    // Create a note.
    var dateCreated = new Date();
    var id = dateCreated.getTime().toString();
    var noteModel = new Notes.NoteModel({
        id: id,
        dateCreated: dateCreated,
        title: "",
        narrative: ""
    });

    Notes.dataContext.init(notesListStorageKey);
    Notes.dataContext.saveNote(noteModel);

    // Should contain a note.
    notesList = $.jStorage.get(notesListStorageKey);

    expect(notesList.length).toBe(1);

    // Clean up
    $.jStorage.deleteKey(notesListStorageKey);
});
```

In this spec, we first empty the local storage container we will use. Then, we save a dummy note by calling *saveNote* on the *dataContext* module. Last, we load the notes list from local storage, and assert that it contains one item.

As the *saveNote* function is missing, the test should fail:



7 specs, 1 failure in 0.011s Finished at Fri Feb 24 2012 22:39:44 GMT-0500 (Eastern Standard Time)

Data context tests

Saves a note to local storage

TypeError: Object #<Object> has no method 'saveNote'

```
TypeError: Object #<Object> has no method 'saveNote
   at [object Object].<anonymous> (http://localhost/J
   at [object Object].execute (http://localhost/JQMSar
   at [object Object].next_ (http://localhost/JQMSamp
```

In the *dataContext* module, let's create *saveNote* as follows:

```
var saveNote = function (noteModel) {

    var found = false;
    var i;

    for (i = 0; i < notesList.length; i += 1) {
        if (notesList[i].id === noteModel.id) {
            notesList[i] = noteModel;
            found = true;
            i = notesList.length;
        }
    }

    if (!found) {
        notesList.splice(0, 0, noteModel);
    }

    saveNotesToLocalStorage();
};
```

This function is straightforward. We start by iterating over the array existing notes. If we find the *id* of the edited note, we "edit" the existing note through an in-place replacement with the supplied note. If don't find the *id*, we simply place the supplied note at the beginning of the array.

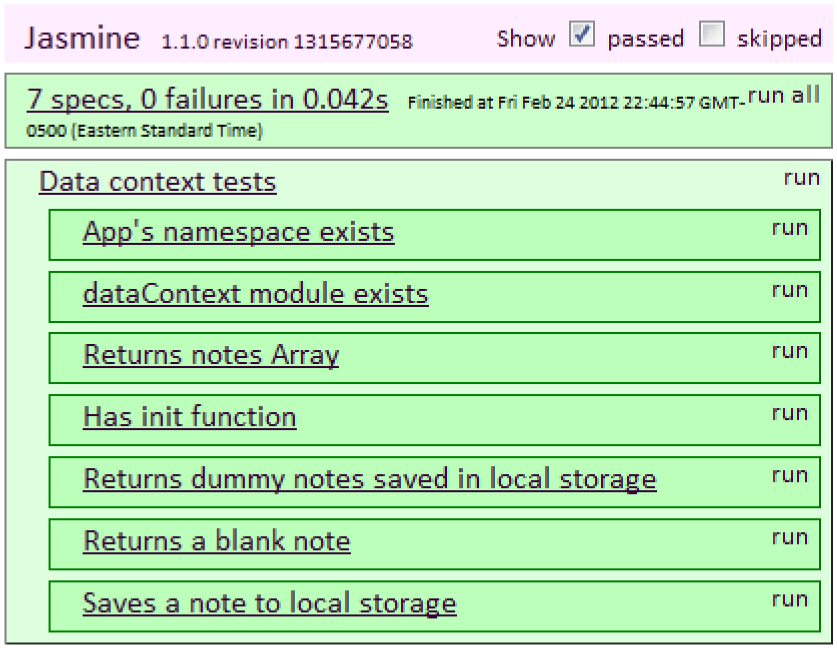
As it is a public function, let's add it to the module's public interface:

```
var public = {
    init: init,
    getNotesList: getNotesList,
    createBlankNote: createBlankNote,
    saveNote: saveNote
};
```

Finally, we call the private function *saveNotesToLocalStorage*, which we also need to add to the *dataContext* module. This is where we save the modified array to local storage:

```
var saveNotesToLocalStorage = function () {
    $.jStorage.set(notesListStorageKey, notesList);
};
```

Let's run the spec again. This time, it should pass:


Jasmine 1.1.0 revision 1315677058 Show ✓ passed ☐ skipped

7 specs, 0 failures in 0.042s Finished at Fri Feb 24 2012 22:44:57 GMT-0500 (Eastern Standard Time) run all

Data context tests run

App's namespace exists run

dataContext module exists run

Returns notes Array run

Has init function run

Returns dummy notes saved in local storage run

Returns a blank note run

Saves a note to local storage run

Back in the *controller*'s *onSaveButtonTapped* function, we also wrote a call to the helper function *returnToNotesListPage*, which we will implement like so:

```
var returnToNotesListPage = function () {

    $.mobile.changePage("#" + notesListPageId,
        { transition: "slide", reverse: true });
};
```

We will call this convenience function every time we need to return to the Notes List page.

There is one additional step we need to implement so we can edit and save notes. We need to make sure we reset the *currentNote* reference after a note is saved. We can do this from the *pagechange* event handler, *onPageChange*, within the *switch* statement:

```
var onPageChange = function (event, data) {

    var toPageId = data.toPage.attr("id");
    var fromPageId = null;

    if (data.options.fromPage) {
        fromPageId = data.options.fromPage.attr("id");
    }

    switch (toPageId) {

        case notesListPageId:
            resetCurrentNote(); // <-- Reset reference to the note being edited.
            renderNotesList();
            break;

        case noteEditorPageId:

            if (fromPageId === notesListPageId) {
                renderSelectedNote(data);
            }
            break;
    }
};
```

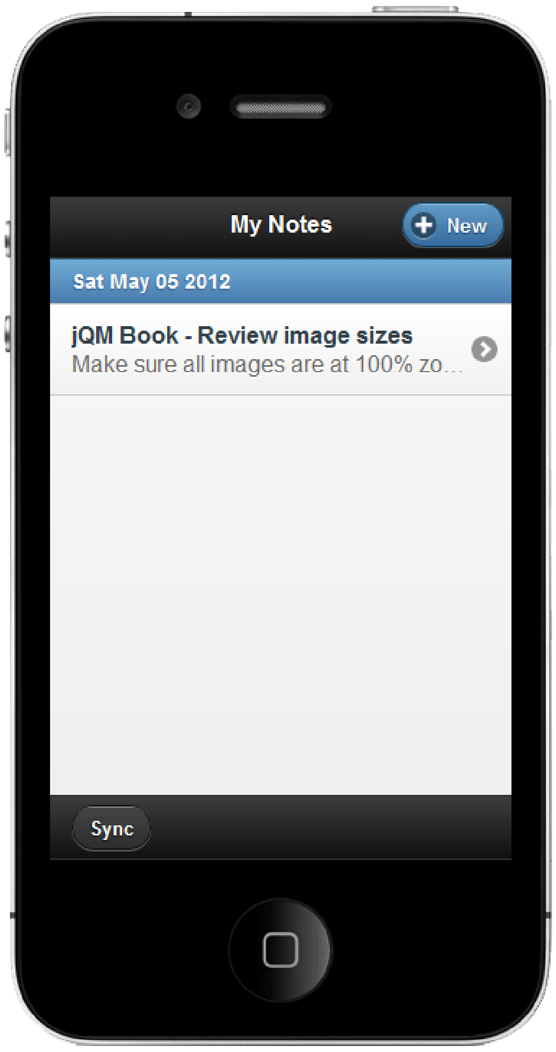
When we're navigating back to the Notes List page, we're now invoking the private function *resetCurrentNote*, which looks like this:

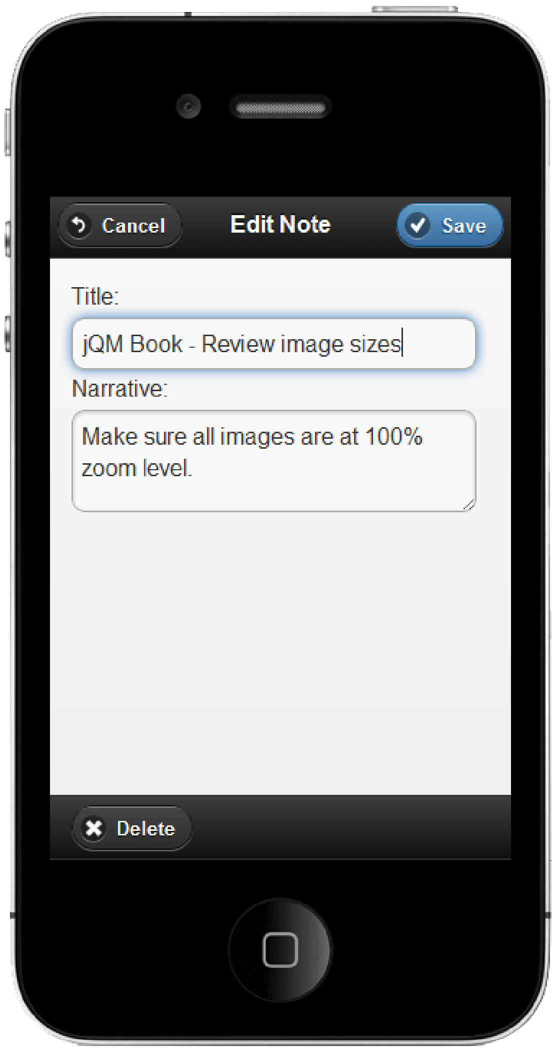
```
var resetCurrentNote = function () {
    currentNote = null;
};
```

We're almost ready to test on the emulator again. First, we'll go back to the *mobileinit* handler and comment out the call to *createDummyNotes*. We don't need it anymore:

```
$(document).bind("mobileinit", function () {
    //Notes.testHelper.createDummyNotes("Notes.NotesList");
    Notes.controller.init();
});
```

Finally, we must clear the emulator's cache to remove the dummy notes we created . Then, check the app on the emulator, where we should be able to create and edit notes:


My Notes
New
Sat May 05 2012
jQM Book - Review image sizes
Make sure all images are at 100% zo...
Sync


Cancel
Edit Note
Save
Title:
jQM Book - Review image sizes
Narrative:
Make sure all images are at 100% zoom level.
Delete

Getting Ready to Validate a Data Model

We decided that we are going to consider that a note is valid when it has a title, and we are not going to force our users to enter a narrative before they can save the note. We defined this behavior through the *isValid* function of the *NoteModel* class:

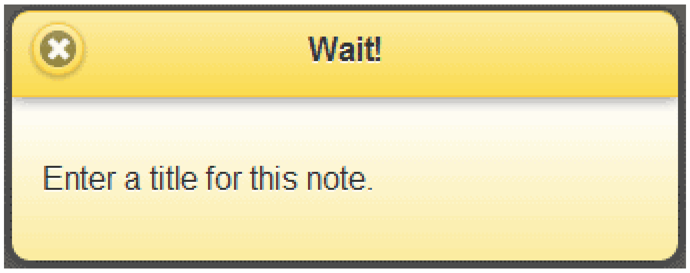
```
Notes.NoteModel.prototype.isValid = function () {
    "use strict";
    if (this.title && this.title.length > 0) {
        return true;
    }
    return false;
};
```

We also left an empty branch in the *onSaveNoteButtonTapped* function of the *controlle*r module, where we need to add the code that will inform our user that her note is invalid.

Here is the function as we originally created it:

```
var onSaveNoteButtonTapped = function () {

    // Validate note.
    var titleEditor = $(noteTitleEditorSel);
    var narrativeEditor = $(noteNarrativeEditorSel);
    var tempNote = dataContext.createBlankNote();


    tempNote.title = titleEditor.val();
    tempNote.narrative = narrativeEditor.val();

    if (tempNote.isValid()) {

        if (null !== currentNote) {

            currentNote.title = tempNote.title;
            currentNote.narrative = tempNote.narrative;
        } else {

            currentNote = tempNote;
        }

        dataContext.saveNote(currentNote);

        returnToNotesListPage();

    } else {
        // TODO: Inform the user the note is invalid.
    }
};
```

Before we complete this function, we need to define what UI elements we will use to inform the user that her note is invalid. We can accomplish this with a jQuery Mobile dialog. In our case, we want to create a very simple dialog, made of a header and a short message to explain our users what is happening:



Enter a title for this note.

How to Create a Dialog With jQuery Mobile

One of the ways you tell jQuery Mobile to display a page as a dialog consists of using the *data-role="dialog"* attribute. We will follow this approach, defining an Invalid Note dialog in the index.html file as follows:

```
<!--Invalid Note dialog-->
<div id="invalid-note-dialog" data-role="dialog" data-title="Invalid Note" data-
theme="e">
    <div data-role="header" data-theme="e">
        <h1>Wait!</h1>
    </div>
    <div data-role="content">
        <p>Enter a title for this note.</p>
    </div>
</div>
```

We will insert this markup right after the Note Editor page in the index.html file. Note how we use the *data-theme="e"* attribute to change the appearance of the dialog. Applying the E swatch to the dialog helps give it more of a warning look.

With the dialog in place, we need to go in the *controller* module and add an identifier for it:

```
var invalidNoteDlgSel = "#invalid-note-dialog";
```

This identifier will allow us to activate the dialog in the *onSaveNoteButtonTapped* method of the *controller* module like so:

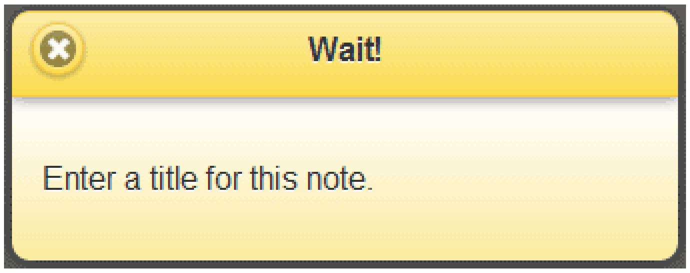
```
var onSaveNoteButtonTapped = function () {

    // Validate note.
    var titleEditor = $(noteTitleEditorSel);
    var narrativeEditor = $(noteNarrativeEditorSel);
    var tempNote = dataContext.createBlankNote();


    tempNote.title = titleEditor.val();
    tempNote.narrative = narrativeEditor.val();
    if (tempNote.isValid()) {

        if (null !== currentNote) {

            currentNote.title = tempNote.title;
            currentNote.narrative = tempNote.narrative;
        } else {

            currentNote = tempNote;
        }

        dataContext.saveNote(currentNote);

        returnToNotesListPage();

    } else {
        // Inform the user the note is invalid.
        $.mobile.changePage(invalidNoteDlgSel, defaultDlgTrsn);
    }
};
```

Note that we are activating the dialog via the *$.mobile.changePage* method, which you can use to trigger page changes programmatically. This function takes a reference to the page in question, as well as the transition you want to use when bringing the page into view.

Instead of passing an inline-defined transition, we are going to define a default transition, which we will use for all the dialogs in the application. The following one-liner will define *defaultDlgTrsn* right at the beginning of the *controller* module:

```
var defaultDlgTrsn = { transition: "slideup" };
```

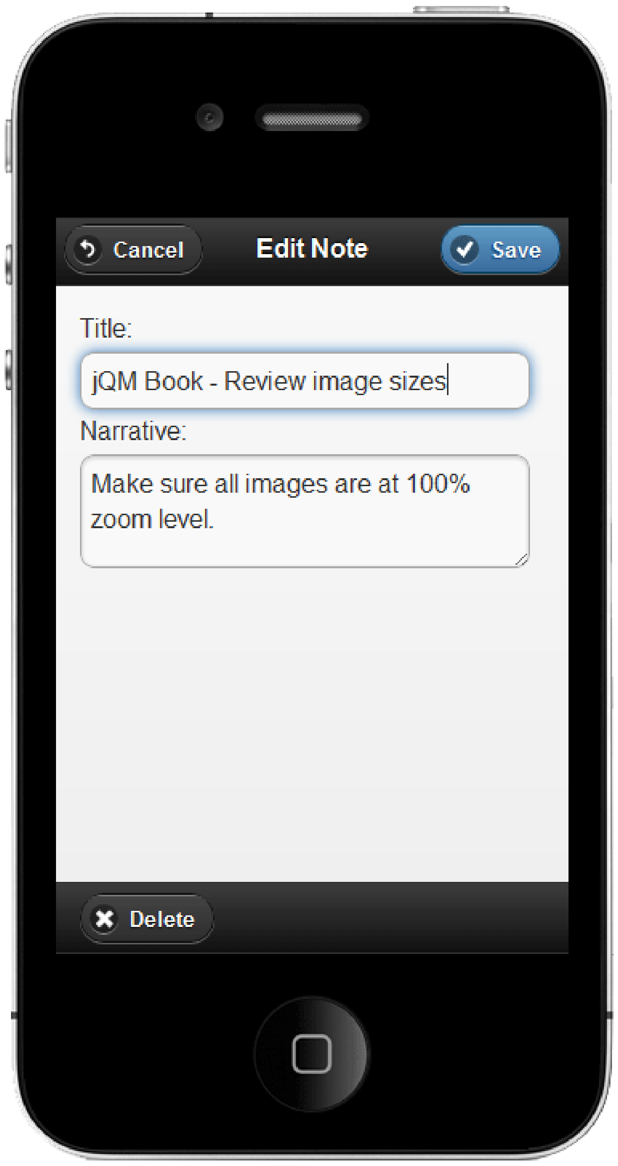
It is time to check how the dialog looks. If we start our emulator and try to save a note with a blank title, we should see the Invalid Note dialog become active:



Enter a title for this note.

Creating a Confirmation Dialog

The last feature we need to address in the Note Editor page is deleting a note. Users of the application will initiate this workflow by tapping the Delete button on the Edit Note page:



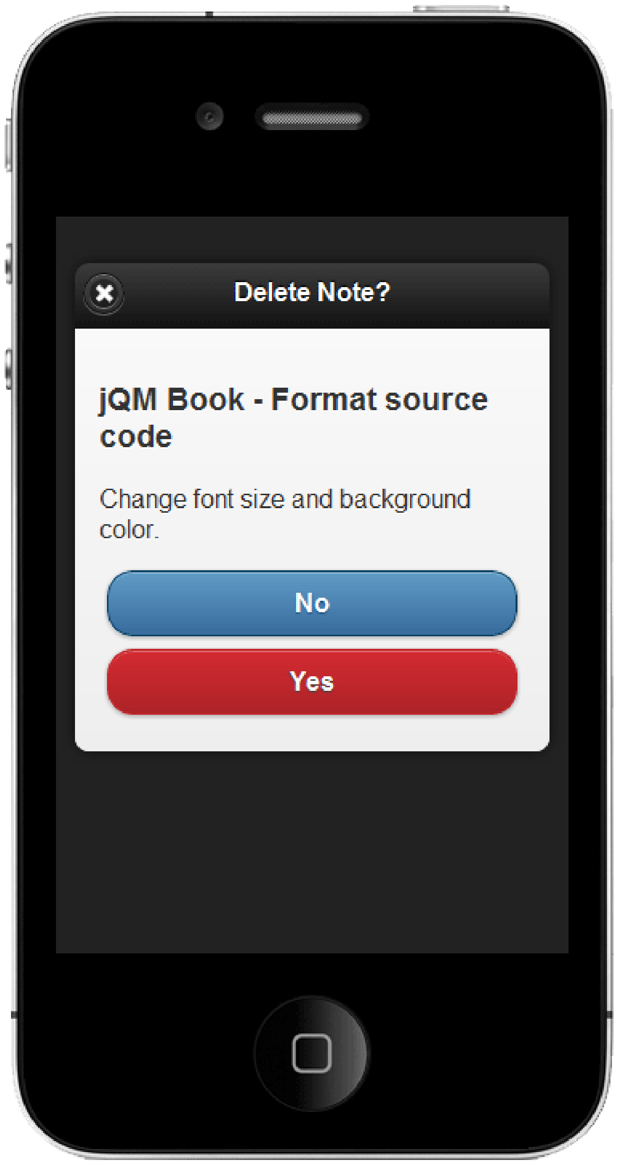
Title:

jQM Book - Review image sizes

Narrative:

Make sure all images are at 100% zoom level.

When the user taps the button, we are going to render a small dialog, asking her for confirmation:


Delete Note?
jQM Book - Format source code
Change font size and background color.
No
Yes

If the user taps the No button on the confirmation dialog, we will just return to the Note Editor page. If she taps the Yes button, we will proceed to delete the note, and then return to the Notes List page.

Let’s first create the dialog, and then connect it to the Delete button on the Note Editor page. We will use the following markup to create the Confirm Delete Note dialog in the index.html file:

```
<!-- Confirm Delete Note dialog-->
<div id="confirm-delete-note-dialog" data-role="dialog" data-title="Delete Note" >
    <div data-role="header">
    <h1>
            Delete Note?</h1>
    </div>
    <div data-role="content">
```

```
        <div id="delete-note-content-placeholder"></div>
        <a id="cancel-delete-note-button" data-role="button" data-theme="b" data-
rel="back">No</a>
        <a id="ok-to-delete-note-button" data-role="button" data-theme="f">Yes</a>
    </div>
</div>
```

Notice how we again use the *data-role="dialog"* attribute to have the Framework render this page as a dialog. We will use the *delete-note-content-placeholder* div to render the selected note. The interesting thing about the buttons, which are links decorated with the *data-role="button"* attribute, is how we're using theme swatches to define their colors.

We're using the B swatch, a built-in swatch of the default jQuery Mobile theme, for the No button. The Yes button uses the F swatch to give the button the red color. This is a custom swatch that we will create in a few minutes.

With the dialog created, we need to focus on how we will render it from within the *controller* module. Evidently, we need to create an identifier for the dialog:

```
var confirmDeleteNoteDlgSel = "#confirm-delete-note-dialog";
```

We also need identifiers for the Delete button in the Note Editor page, the Yes button in the Confirm Delete Note dialog, and the div element that will serve as placeholder in the dialog. Let's add them to the *controller* module like so:

```
var deleteNoteButtonSel = "#delete-note-button",
    deleteNoteContentPlaceholderSel = "#delete-note-content-placeholder",
    okToDeleteNoteButtonSel = "#ok-to-delete-note-button";
```

As the dialog will be activated upon the user tapping the Delete button in the Note Editor page, we will define a *tap* handler for this button in the *init* function of the controller. Something like this will do:

```
var init = function () {

    // Rest of the function omitted for brevity.

    d.delegate(deleteNoteButtonSel, "tap", onDeleteNoteButtonTapped);
};
```

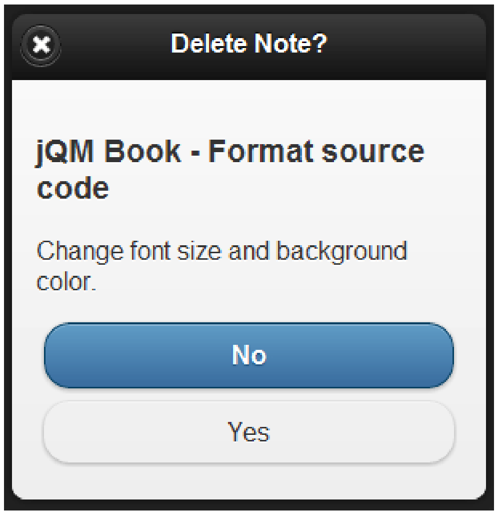
Here we are saying that the tap event on the button will invoke the *onDeleteNoteButtonTapped* function, which we will now add to the *controller* module:

```
var onDeleteNoteButtonTapped = function () {

    if (currentNote) {
        // Render selected note in confirmation dlg.
        // Deletion will be handled elsewhere, after user confirms it's ok to delete.

        var noteContentPlaceholder = $(deleteNoteContentPlaceholderSel);

        noteContentPlaceholder.empty();
        $("<h3>" + currentNote.title + "</h3><p>" + currentNote.narrative +
"</p>").appendTo(noteContentPlaceholder);

        $.mobile.changePage(confirmDeleteNoteDlgSel, defaultDlgTrsn);
    }
};
```

In *onDeleteNotebuttonTapped*, we first render the current note's title and narrative in the content area of the dialog. We can accomplish this by simply appending html nodes to the placeholder we defined within the dialog. Then, we use the *$mobile.changePage* function to make the dialog the active page.

Note how re-use the *defaultDlgTrsn*, previously defined when we were creating the Invalid Note dialog.

Now we are at a point where we are waiting for the user's answer to our question – delete the note, yes or no? The answer will tell us whether to delete the note or cancel the workflow.


Delete Note?
jQM Book - Format source code
Change font size and background color.
No
Yes

Deleting a Note

Cancellation will occur upon the user tapping the No button. We don't have to write code for this scenario, as we are using the *data-rel="back"* attribute for the No button:

```
<a id="cancel-delete-note-button" data-role="button" data-theme="b" data-
rel="back">No</a>
```

This role will cause the button's *tap* event to initiate a transition to the previous page, the Note Editor page, which is exactly what we want.

The Yes button is a little different. When the user taps this button, we need to delete the note and initiate a transition to the Notes List page. The Notes List page should then render the updated notes list.

We are going to define the *tap* handler for the Yes button in the controller module's *init* function, binding to the button's *tap* event:

```
var init = function () {

    // Rest of the function omitted for brevity.

    d.delegate(okToDeleteNoteButtonSel, "tap", onOKToDeleteNoteButtonTapped);
};
```

Then, we will define *onOKToDeleteNoteButtonTapped* like so:

```
var onOKToDeleteNoteButtonTapped = function () {

    dataContext.deleteNote(currentNote);
    returnToNotesListPage();
};
```

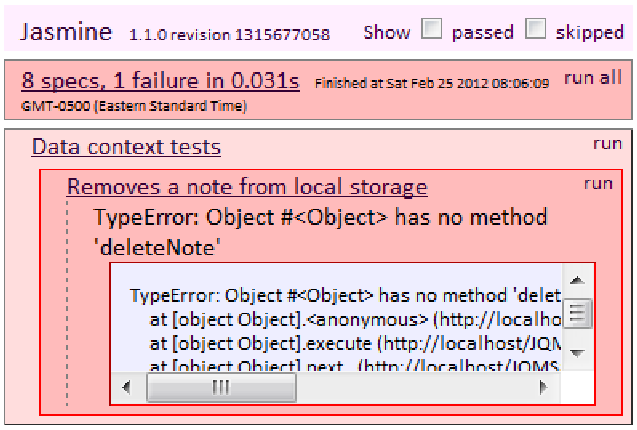
Simple, right? We first call the *dataContext* module's *deleteNote*, and then transition to the Notes List page by calling *returnToNotesListPage*, a function that we created when we were working on the steps required to save a note.

Before defining the *deleteNote* function in the *dataContext* module, let's create a test for it in the AppSpec.js file:

```
it("Removes a note from local storage", function () {

    // Create a note.
    var dateCreated = new Date();
    var id = dateCreated.getTime().toString();

    var noteModel = new Notes.NoteModel({
        id: id,
        dateCreated: dateCreated,
        title: "",
        narrative: ""
    });

    // Start with an empty notes list.
    var notesList = [];
    // Add note to local storage.
    notesList.push(noteModel);
    $.jStorage.set(notesListStorageKey, notesList);
    notesList = $.jStorage.get(notesListStorageKey);
    expect(notesList.length).toEqual(1);

    // Proceed to delete.
    Notes.dataContext.init(notesListStorageKey);
    Notes.dataContext.deleteNote(noteModel);

    // Should retrieve empty array
    notesList = $.jStorage.get(notesListStorageKey);
    expect(notesList.length).toEqual(0);

    // Clean up
    $.jStorage.deleteKey(notesListStorageKey);

});
```

In the test, we first create a note and save it directly to local storage. Then, we use the function being tested, *deleteNote*, to remove the note. The expectation is that we can delete a note using this function.

The test should fail, as *deleteNote* still does not exist:



```
Jasmine 1.1.0 revision 1315677058     Show ☐ passed ☐ skipped
8 specs, 1 failure in 0.031s  Finished at Sat Feb 25 2012 08:06:09  run all
GMT-0500 (Eastern Standard Time)
Data context tests                                              run
  Removes a note from local storage                             run
    TypeError: Object #<Object> has no method
    'deleteNote'
      TypeError: Object #<Object> has no method 'delet
        at [object Object].<anonymous> (http://localho
        at [object Object].execute (http://localhost/JQN
        at [object Object].next_ (http://localhost/JQMS
```

With the test in place, let's head over to the DataContext.js file, and define *deleteNote* like so:

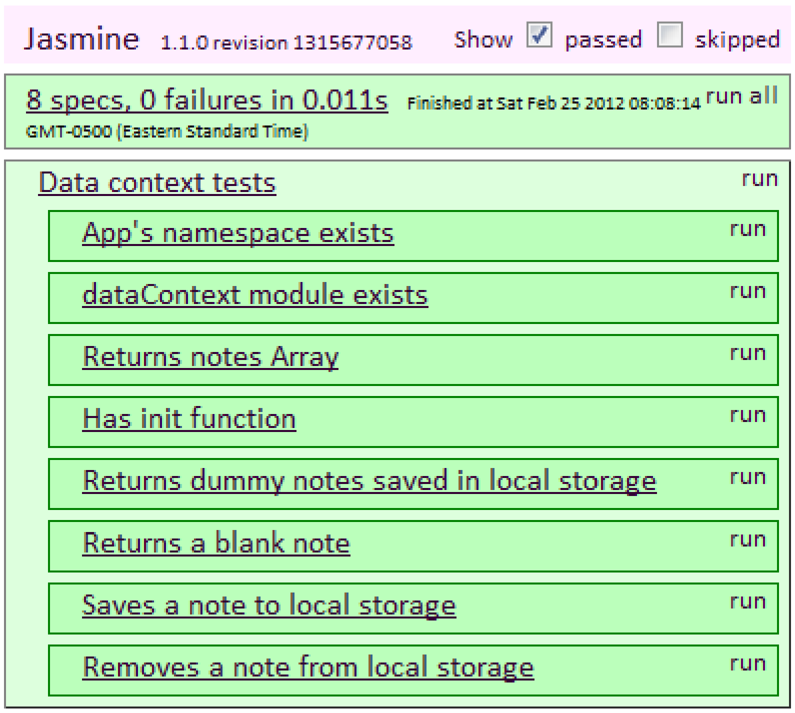
```
var deleteNote = function (noteModel) {

    var i;
    for (i = 0; i < notesList.length; i += 1) {
        if (notesList[i].id === noteModel.id) {
            notesList.splice(i, 1);
            i = notesList.length;
        }
    }

    saveNotesToLocalStorage();
};
```

Again, a very simple function that loops through the array of existing notes, trying to find one with the id of the note we want to delete. If found, the note is removed from the array. The updated array is then serialized to local storage through a call to *saveNotesToLocalStorage*.

We also need to add *deleteNote* to the public interface of the module so we can invoke it from the *controller* module:

```
var public = {
    init: init,
    getNotesList: getNotesList,
    createBlankNote: createBlankNote,
    saveNote: saveNote,
    deleteNote: deleteNote
};
```

Time to re-run the test, which should pass if we didn't make any mistakes:


Jasmine 1.1.0 revision 1315677058 Show ☑ passed ☐ skipped

8 specs, 0 failures in 0.011s Finished at Sat Feb 25 2012 08:08:14 GMT-0500 (Eastern Standard Time) run all

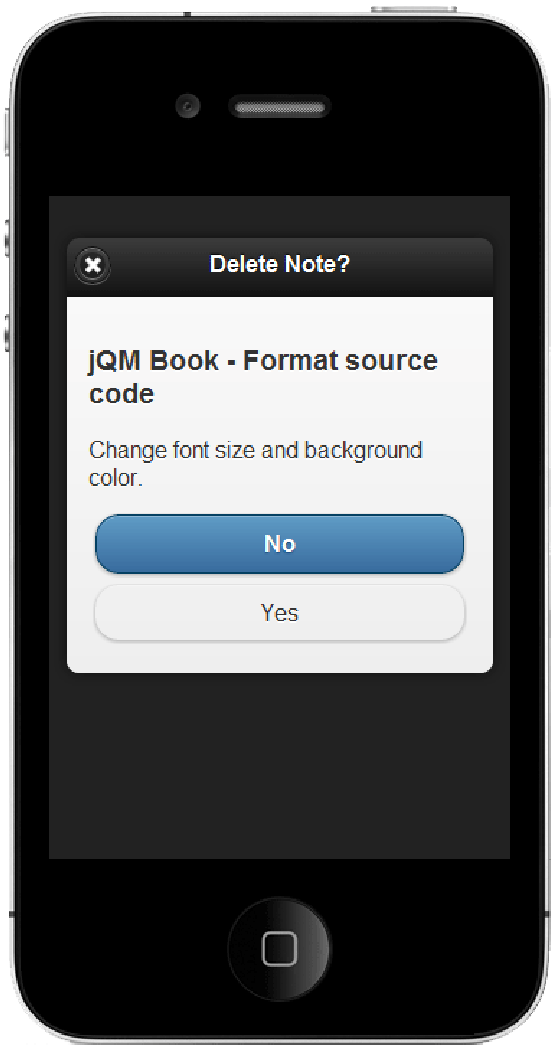
Data context tests run
- App's namespace exists run
- dataContext module exists run
- Returns notes Array run
- Has init function run
- Returns dummy notes saved in local storage run
- Returns a blank note run
- Saves a note to local storage run
- Removes a note from local storage run

This completes the code that we needed for the Delete Note feature. The UI elements are in place, and the *controller* and *dataContext* modules are ready to handle this workflow.

Now you can fire up the emulator, and verify that you can delete notes.

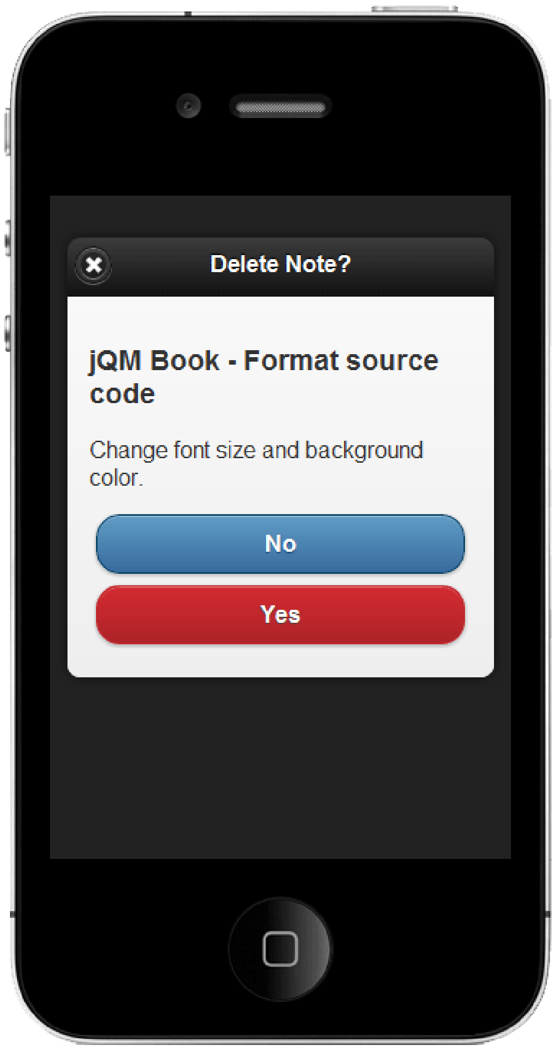
Using a Custom Theme Swatch in jQuery Mobile

Before we end this chapter, let us take care of a cosmetic issue. How about changing the color of the Yes button in the Delete Note dialog?



jQM Book - Format source code

Change font size and background color.



jQM Book - Format source code

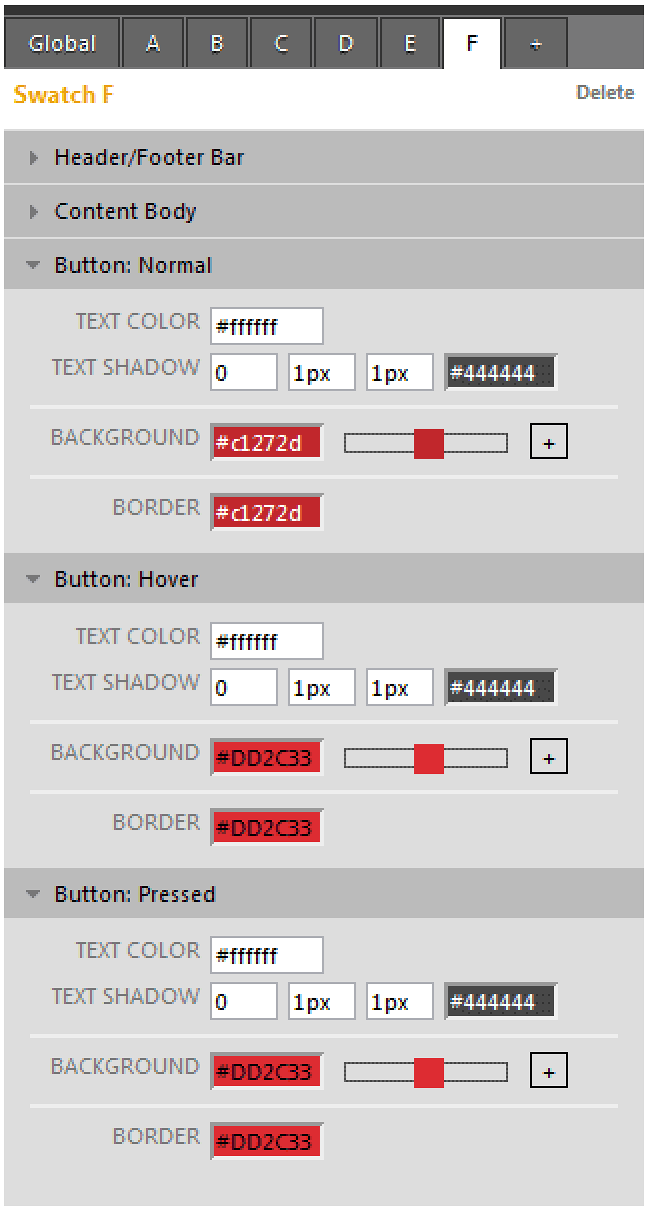
Change font size and background color.

One approach to accomplish this consists of using a custom jQuery Mobile theme swatch. We will take advantage of the ThemeRoller for jQuery Mobile (http://jquerymobile.com/themeroller/) to change the swatch of the Yes button.

As you already saw, we assigned the *data-theme="f"* attribute to the button when we created the dialog:

```
<a id="ok-to-delete-note-button" data-role="button" data-theme="f">Yes</a>
```

With this in mind, we will head over to the ThemeRoller website and define an F swatch. The swatch has several properties, but we're only interested in those that apply to buttons. Let's use the following properties for the different buttons states:



Button: Normal

TEXT COLOR #ffffff
TEXT SHADOW 0 1px 1px #444444
BACKGROUND #c1272d
BORDER #c1272d

Button: Hover

TEXT COLOR #ffffff
TEXT SHADOW 0 1px 1px #444444
BACKGROUND #DD2C33
BORDER #DD2C33

Button: Pressed

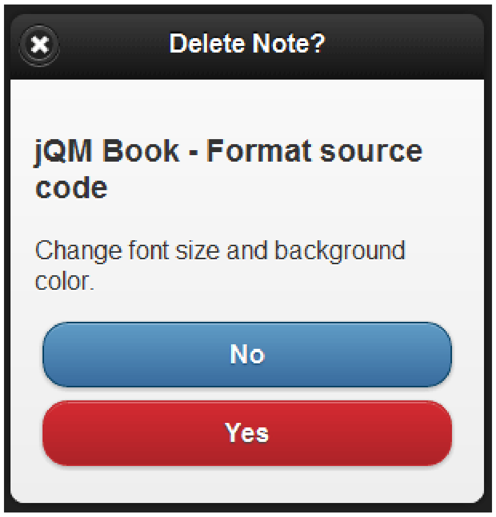
TEXT COLOR #ffffff
TEXT SHADOW 0 1px 1px #444444
BACKGROUND #DD2C33
BORDER #DD2C33

After entering these values, we can use the Download Theme link to download the theme file.

The file contains styles for all elements enhanced by the jQuery Mobile Framework. We will copy the styles that apply to button elements, or elements decorated with the *data-role="button"* attribute, which we will add to our app.css file like so:

```
.ui-btn-up-f {
	border: 1px solid #c1272d /*{f-bup-border}*/;
	background: #c1272d /*{f-bup-background-color}*/;
	font-weight: bold;
	color: #ffffff /*{f-bup-color}*/;
	text-shadow:  0  /*{f-bup-shadow-x}*/  1px  /*{f-bup-shadow-y}*/  1px  /*{f-
bup-shadow-radius}*/ #444444 /*{f-bup-shadow-color}*/;
	background-image: -webkit-gradient(linear, left top, left bottom, from(
#D42A31 /*{f-bup-background-start}*/), to( #AD2328 /*{f-bup-background-end}*/)); /*
Saf4+, Chrome */
	background-image: -webkit-linear-gradient(top, #D42A31 /*{f-bup-background-
start}*/, #AD2328 /*{f-bup-background-end}*/); /* Chrome 10+, Saf5.1+ */
	background-image:    -moz-linear-gradient(top, #D42A31 /*{f-bup-background-
start}*/, #AD2328 /*{f-bup-background-end}*/); /* FF3.6 */
	background-image:     -ms-linear-gradient(top, #D42A31 /*{f-bup-background-
start}*/, #AD2328 /*{f-bup-background-end}*/); /* IE10 */
	background-image:      -o-linear-gradient(top, #D42A31 /*{f-bup-background-
start}*/, #AD2328 /*{f-bup-background-end}*/); /* Opera 11.10+ */
	background-image:         linear-gradient(top, #D42A31 /*{f-bup-background-
start}*/, #AD2328 /*{f-bup-background-end}*/);
}
.ui-btn-up-f a.ui-link-inherit {
	color: #ffffff /*{f-bup-color}*/;
}

.ui-btn-hover-f {
	border: 1px solid #DD2C33 /*{f-bhover-border}*/;
	background: #DD2C33 /*{f-bhover-background-color}*/;
	font-weight: bold;
	color: #ffffff /*{f-bhover-color}*/;
	text-shadow:  0  /*{f-bhover-shadow-x}*/  1px  /*{f-bhover-shadow-y}*/  1px
/*{f-bhover-shadow-radius}*/ #444444 /*{f-bhover-shadow-color}*/;
	background-image: -webkit-gradient(linear, left top, left bottom, from(
#F33038 /*{f-bhover-background-start}*/), to( #C6272D /*{f-bhover-background-
end}*/)); /* Saf4+, Chrome */
	background-image: -webkit-linear-gradient(top, #F33038 /*{f-bhover-background-
start}*/, #C6272D /*{f-bhover-background-end}*/); /* Chrome 10+, Saf5.1+ */
	background-image:    -moz-linear-gradient(top, #F33038 /*{f-bhover-background-
start}*/, #C6272D /*{f-bhover-background-end}*/); /* FF3.6 */
	background-image:     -ms-linear-gradient(top, #F33038 /*{f-bhover-background-
start}*/, #C6272D /*{f-bhover-background-end}*/); /* IE10 */
	background-image:      -o-linear-gradient(top, #F33038 /*{f-bhover-background-
start}*/, #C6272D /*{f-bhover-background-end}*/); /* Opera 11.10+ */
	background-image:         linear-gradient(top, #F33038 /*{f-bhover-background-
start}*/, #C6272D /*{f-bhover-background-end}*/);
}
.ui-btn-hover-f a.ui-link-inherit {
	color: #ffffff /*{f-bhover-color}*/;
}
.ui-btn-down-f {
	border: 1px solid #DD2C33 /*{f-bdown-border}*/;
	background: #DD2C33 /*{f-bdown-background-color}*/;
	font-weight: bold;
	color: #ffffff /*{f-bdown-color}*/;
	text-shadow:  0  /*{f-bdown-shadow-x}*/  1px  /*{f-bdown-shadow-y}*/  1px
/*{f-bdown-shadow-radius}*/ #444444 /*{f-bdown-shadow-color}*/;
	background-image: -webkit-gradient(linear, left top, left bottom, from(
#C6272D /*{f-bdown-background-start}*/), to( #F33038 /*{f-bdown-background-end}*/));
/* Saf4+, Chrome */
	background-image: -webkit-linear-gradient(top, #C6272D /*{f-bdown-background-
start}*/, #F33038 /*{f-bdown-background-end}*/); /* Chrome 10+, Saf5.1+ */
	background-image:    -moz-linear-gradient(top, #C6272D /*{f-bdown-background-
start}*/, #F33038 /*{f-bdown-background-end}*/); /* FF3.6 */
	background-image:     -ms-linear-gradient(top, #C6272D /*{f-bdown-background-
start}*/, #F33038 /*{f-bdown-background-end}*/); /* IE10 */
	background-image:      -o-linear-gradient(top, #C6272D /*{f-bdown-background-
start}*/, #F33038 /*{f-bdown-background-end}*/); /* Opera 11.10+ */
	background-image:         linear-gradient(top, #C6272D /*{f-bdown-background-
start}*/, #F33038 /*{f-bdown-background-end}*/);
}
.ui-btn-down-f a.ui-link-inherit {
	color: #ffffff /*{f-bdown-color}*/;
}
```

After adding these classes to the app.css file, we can check the dialog's look. The Yes button should now render with the new F swatch's properties:


Delete Note?
jQM Book - Format source code
Change font size and background color.
No
Yes

Where Are We?

In this chapter we learned how to create a jQuery Mobile form, how load data into the form, and how to save form data to local storage. We also learned how to create jQuery Mobile dialogs, and how to create a custom jQuery Mobile theme swatch.

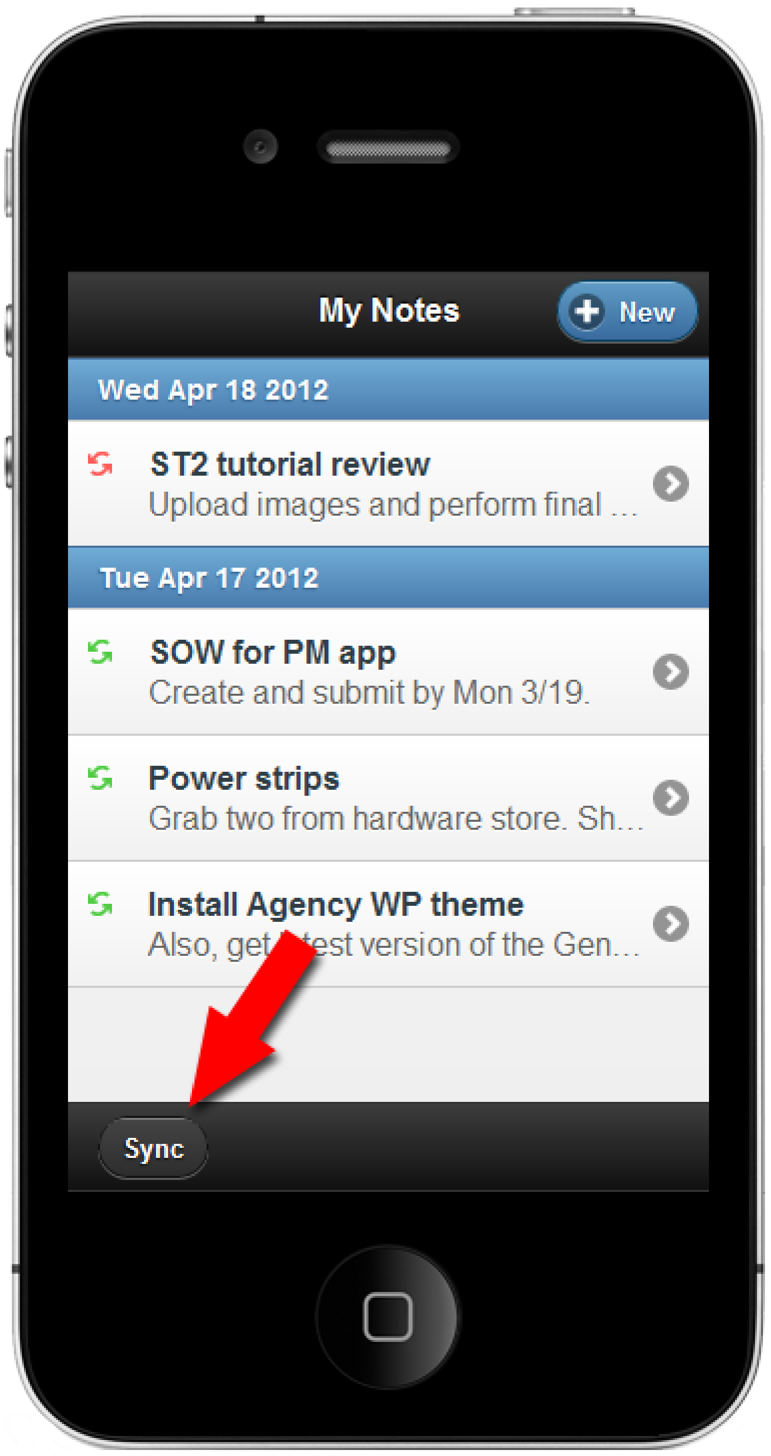
We are very close to having a feature-complete application, as the app allows us to create, update and delete notes.

Next, we are going to learn how to synchronize the notes cache with the server.

Chapter 5: Synchronizing With the Server

What You Will Learn In This Chapter

In this chapter we are going to implement one important feature of the application – Keeping the list of notes synchronized with the server and other clients. Users will have access to this feature through the Sync button we placed in the Notes List Page:


My Notes
New
Wed Apr 18 2012
ST2 tutorial review
Upload images and perform final …
Tue Apr 17 2012
SOW for PM app
Create and submit by Mon 3/19.
Power strips
Grab two from hardware store. Sh…
Install Agency WP theme
Also, get latest version of the Gen…
Sync

While building the synchronization feature, we will cover the following topics:

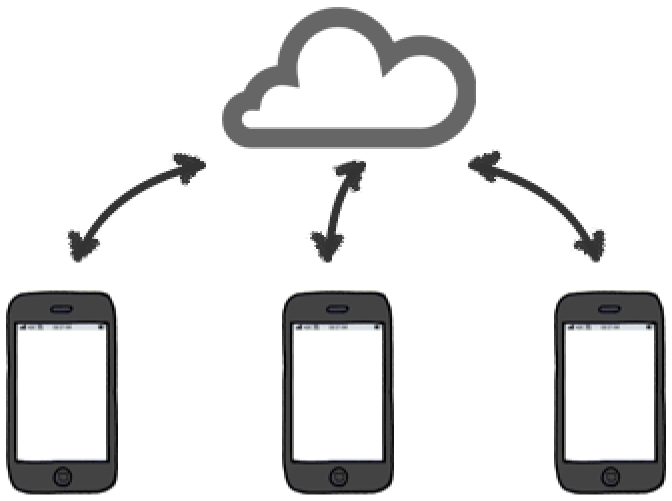
- How to upload data to a server.
- How to receive data from a server.
- How to keep application data synchronized with the server.

How Synchronization Will Work

Let's take a minute to think about what needs to happen to keep the notes in sync.

We are going to design this feature assuming a hypothetical scenario where a number of devices running the application will keep a list of notes synchronized. Synchronization will occur through the server.

In the application, when a user taps the Sync button, we want to send the list of cached notes to the server so they can be safely stored. We also want to receive new notes that other clients have stored on the server. In short, we are going to implement two-way synchronization.



As we have done with all the previous data access routines, we will place the synchronization logic in the *dataContext* module. We also want to know when the synchronization is finished so we can refresh the notes list presented to the user. We will accomplish this through a callback function, which we will invoke once the *dataContext* module completes the sync operation.

We want the *tap* event on the Sync button to invoke a handler in the *controller* module. This will in turn invoke the synchronization function in the *dataContext* module.

Creating a Tap Handler for the Sync Button

We will first work on the *tap* event handler for the Sync button. In the *controller* module, we are going to create a reference to the Sync button like so:

```
var syncNotesWithServerBtnSel = "#sync-notes-button";
```

Next, we will hook up a *tap* handler for the button. We can do this in the controller's *init* function:

```
var init = function () {

    // Rest of the init() function omitted for brevity...


    d.delegate(syncNotesWithServerBtnSel, "tap", onSyncNotesWithServerButtonTapped);
};
```

Now, the *onSyncNotesWithServerButtonTapped* function, which we will add to the controller module, right before the *init* function:

```
var onSyncNotesWithServerButtonTapped = function () {

    dataContext.startNotesSync(onNotesSyncCompleted);
};
```

The handler is simply delegating the job of synchronizing notes to the *dataContext* module's *startNotesSync* function, which takes the callback function *onNotesSyncCompleted* as an argument. We will define this callback in a few minutes.

Changing jQuery Mobile's Default Loading Message

Uploads and downloads from the server do not happen instantaneously. Before we continue our work in the *dataContext* module, we are going to add two helper variables to the *controller* module. They will allow us to alert the user when the synch operation is in progress.

Let's define the *syncingMsg* and *defaultLoadingMsg* variables like so:

```
var defaultLoadingMsg,
    syncingMsg = "Syncing Notes";
```

The purpose of *syncingMsg* is obvious. We will use this variable to store the message we want to display when the synchronization operation is in progress. The *defaultLoadingMsg* variable will temporarily store jQuery Mobile's default loading message while we display the "Syncing Notes" message.

We will store the default *loadingMessage* value by adding this line to the *init* function in the *controller* module:

```
defaultLoadingMsg = $.mobile.loadingMessage;
```

Now we will modify the *onSyncNotesWithServerButtonTapped* function as follows so it displays the “Syncing Notes” message:

```
var onSyncNotesWithServerButtonTapped = function () {

    $.mobile.loadingMessage = syncingMsg;
    $.mobile.showPageLoadingMsg();
    dataContext.startNotesSync(onNotesSyncCompleted);
};
```

In the above code, we are invoking the Framework’s *showPageLoadingMsg* function after setting the loading message to the value we desire. This takes care of showing the correct message to the user. When the *onNotesSyncCompleted* callback is invoked, we will need to turn the message off, and restore the default value of the Framework’s *loadingMessage*.

The *onNotesSyncCompleted* callback is a function that will first re-render the notes list, and then turn off the “Syncing Notes” message:

```
var onNotesSyncCompleted = function (syncResult) {

    renderNotesList();
    $.mobile.hidePageLoadingMsg();
    $.mobile.loadingMessage = defaultLoadingMsg;

};
```

This is all it takes from the *controller* module’s perspective to handle the notes synchronization feature. The more interesting pieces are in the *dataContext* module.

Creating a Test for An Asynchronous Operation

After a tap on the Sync Button, the *controller* will ask the *dataContext* module to synchronize the existing notes with the server:

```
var onSyncNotesWithServerButtonTapped = function () {

    $.mobile.loadingMessage = syncingMsg;
    $.mobile.showPageLoadingMsg();
    dataContext.startNotesSync(onNotesSyncCompleted);
};
```

Let’s first define an empty *startNotesSync* function in the *dataContext* module:

```
var startNotesSync = function (callback) {


};
```

We are also going to expose this function through the module’s public interface:

```
var pub = {
    // Other public functions omitted for brevity.

    startNotesSync: startNotesSync
};
```

The *startNotesSync* function will initiate an AJAX request to the server, uploading the notes cached on the device, and downloading the notes that exist on the server. Before we implement this function, we are going to define the following test in the AppSpec.js file:

```
it("Invokes callback after notes sync with server", function () {

    var notesSyncTestDelayInMs = 15000; // 15 sec. Increase this delay if the test
fails because the server call has not returned.
    var callBackInvoked = false;

    var callback = function (syncResult) {
        callBackInvoked = true;
    };

    runs(function () {

        Notes.dataContext.init({
            storageKey: notesListStorageKey,
            serverUrl: serverUrlForTest
        });
        Notes.dataContext.startNotesSync(callback);

    });

    waits(notesSyncTestDelayInMs);

    runs(function () {
        expect(callBackInvoked).toBeTruthy();
    });

});
```

The purpose of this test is to make sure that the *startNotesSync* function effectively invokes the supplied callback. Invoking the callback will notify the *controller* module that the synchronization finished. Let's take a detailed look at the test, as in it we are using features of the Jasmine Framework that we had not used before.

We begin the test creating a couple of variables and a simple callback function that will help us perform the test:

```
var notesSyncTestDelayInMs = 15000;
var callBackInvoked = false;

var callback = function (syncResult) {
    callBackInvoked = true;
};
```

Next, we use Jasmine's *runs* function to define a couple of blocks of statements that will run serially. The first block initializes the *dataContext* module and invokes the *startNotesSync* function:

```
// First runs() block.
runs(function () {

    Notes.dataContext.init({
        storageKey: notesListStorageKey,
        serverUrl: serverUrlForTest
    });
    Notes.dataContext.startNotesSync(callback);

});
```

The second block invokes the expectation. This is where we verify that the callback was invoked:

```
// Second runs() block.
runs(function () {
    expect(callBackInvoked).toBeTruthy();
});
```

Jasmine's *waits* function introduces a delay between the two *runs* blocks:

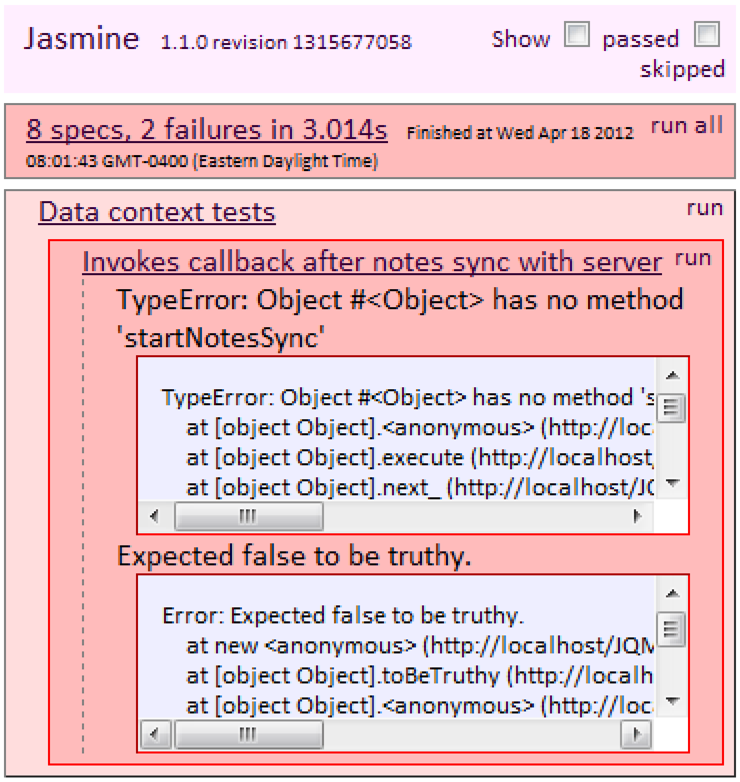
```
waits(notesSyncTestDelayInMs);
```

The delay allows us to wait for the operations started by *startNotesSync* to finish before we proceed to confirm that the callback was invoked. This is a very convenient way to test long-running operations. For example, operations that perform AJAX requests.

Before running the test, we need to define the *serverUrlForTest* variable at the top of the test suite:

```
var notesListStorageKey = "Notes.NotesListTest",
    serverUrlForTest = "notes.aspx";
```

If we go ahead and run the test, it should fail:



8 specs, 2 failures in 3.014s Finished at Wed Apr 18 2012 08:01:43 GMT-0400 (Eastern Daylight Time)

Data context tests

Invokes callback after notes sync with server

TypeError: Object #<Object> has no method 'startNotesSync'

```
TypeError: Object #<Object> has no method 's
    at [object Object].<anonymous> (http://loc
    at [object Object].execute (http://localhost
    at [object Object].next_ (http://localhost/JC
```

Expected false to be truthy.

```
Error: Expected false to be truthy.
    at new <anonymous> (http://localhost/JQM
    at [object Object].toBeTruthy (http://localh
    at [object Object].<anonymous> (http://loc
```

There is nothing wrong with this. Remember that we are missing the *startNotesSync* implementation.

Passing a Server URL to the dataContext Module

Before we continue with *startNotesSync*, let's make a small modification to the *dataContext* module. First, we will add the *serverUrl* variable to the module. This variable will store the URL the module will use for notes synchronization:

```
var serverUrl;
```

Then, we are going to modify the *init* function, replacing the *storageKey* argument with a more generic *config* argument, which will allow us pass not only the *storageKey*, but also the *serverUrl* value:

```
 var init = function (config) {
    notesListStorageKey = config.storageKey;
    serverUrl = config.serverUrl;
    loadNotesFromLocalStorage();
};
```

This is all we need to pass the server's URL to this module. You probably noticed that we used this configuration in the asynchronous operation test we created in the previous section.

Now we need to revisit the *controller* module's *init* function, and make sure we pass the *config* object to the *dataContext*'s own *init*. Let's open the Controller.js file and modify the *init* function as follows:

```
var init = function () {

    dataContext.init({
        storageKey: appStorageKey,
        serverUrl: appServerUrl
    });

    var d = $(document);
    d.bind("pagebeforechange", onPageBeforeChange);
    d.bind("pagechange", onPageChange);
    d.delegate(saveNoteButtonSel, "tap", onSaveNoteButtonTapped);
    d.delegate(deleteNoteButtonSel, "tap", onDeleteNoteButtonTapped);
    d.delegate(okToDeleteNoteButtonSel, "tap", onOKToDeleteNoteButtonTapped);
    d.delegate(syncNotesWithServerBtnSel, "tap", onSyncNotesWithServerButtonTapped);

    defaultLoadingMsg = $.mobile.loadingMessage;
};
```

Now, the *controller*'s *init* is receiving an object containing the *storageKey* and *serverUrl* values needed by the *dataContext*'s *init* function. Notice that we are assigning the *appServerUrl* value to the *serverUrl* property of the object we pass to the *init* function. Let's go ahead and define *appServerUrl* at the top of the *controller* module:

```
var appServerUrl = "notes.php";
```

We will initially perform the notes synchronization against a PHP page. We will write the code for this page at the end of this chapter, along with a C# version.

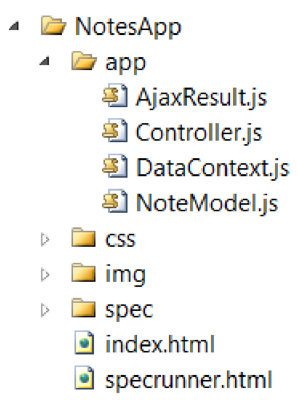
Beginning the Notes Synchronization

Our synchronization workflow will begin with a tap of the Sync Button, which will invoke the *startNotesSync* function. We are going to implement *startNotesSync* like so:

```
var startNotesSync = function (callback) {
    $.ajax({
        url: serverUrl,
        type: "post",
        data: "notes=" + JSON.stringify(notesList),
        success: function (data, textStatus, jqXHR) {

            var syncResult = new Notes.AjaxResult({
                success: true,
                errorThrown: null
            });

            endNotesSync(syncResult, data, callback);
        },
        error: function (jqXHR, textStatus, errorThrown) {


            var syncResult = new Notes.AjaxResult({
                success: false,
                errorThrown: errorThrown
            });

            endNotesSync(syncResult, null, callback);
        }
    });
};
```

This function takes a callback passed from the *controller* module as its argument, and fires an AJAX request that sends the serialized notes list to the server. The interesting details lie in the *success* and *error* callbacks. Both callbacks create an instance of the *AjaxResult* class, which we have not defined yet, and then invoke the *endNotesSync* function, which we have not created either.

Let's talk about the *AjaxResult* class. We will use an instance of this class to pass the result of the AJAX request, as well as any error that occurred during the operation, to the *endNotesSync* function. In order to define this class, we are going to create the AjaxResult.js file in the app directory:


NotesApp
app
AjaxResult.js
Controller.js
DataContext.js
NoteModel.js
css
img
spec
index.html
specrunner.html

In the file, we will enter the following code:

```
Notes.AjaxResult = function (config) {
    "use strict"
    this.success = config.success,
    this.errorThrown = config.errorThrown
};
```

As we already discussed, we will just use a couple of properties, *success* and *errorThrown*, to indicate whether the operation succeeded, and store the exception thrown when the operation does not succeed.

Before moving on, we need to include the AjaxResult.js in the index.html and specrunner.html files. We will include it in index.html like so:

```
<head>
    <title></title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link href="../lib/jqm/jquery.mobile-1.3.0.min.css" rel="stylesheet"
type="text/css" />
    <link href="css/app.css" rel="stylesheet" type="text/css" />
    <script src="../lib/jqm/jquery-1.8.2.min.js" type="text/javascript"></script>
    <script src="../lib/jstorage/jstorage.min.js" type="text/javascript"></script>
    <script src="app/DataContext.js" type="text/javascript"></script>
    <script src="app/Controller.js" type="text/javascript"></script>
    <script src="app/NoteModel.js" type="text/javascript"></script>
    <script src="app/AjaxResult.js" type="text/javascript"></script>
    <script src="spec/TestHelper.js" type="text/javascript"></script>
    <script src="../lib/jqm/jquery.mobile-1.3.0.min.js"
type="text/javascript"></script>
</head>
```

Then, in specrunner.html:

```
<head>
    <title>Jasmine Test Runner</title>
    <!-- Libraries -->
    <script src="../lib/jqm/jquery-1.8.2.min.js" type="text/javascript"></script>
    <script src="../lib/jstorage/jstorage.min.js" type="text/javascript"></script>
    <!-- Jasmine -->
    <link href="../lib/jasmine/jasmine.css" rel="stylesheet" type="text/css" />
    <script src="../lib/jasmine/jasmine.js" type="text/javascript"></script>
    <script src="../lib/jasmine/jasmine-html.js" type="text/javascript"></script>
    <!-- App -->
    <script src="app/DataContext.js" type="text/javascript"></script>
    <script src="app/NoteModel.js" type="text/javascript"></script>
    <script src="app/AjaxResult.js" type="text/javascript"></script>
    <!-- Test Helper -->
    <script src="spec/TestHelper.js" type="text/javascript"></script>
    <!-- Spec -->
    <script src="spec/AppSpec.js" type="text/javascript"></script>
</head>
```

Adding Synchronization Time Stamps to the Note Model

If we upload a notes list to the server, and receive a new list, we will not know which notes were updated. The reason for this is that our *NoteModel* class does not have any properties that help us determine if a note was ever uploaded, and, if so, when the upload took place.

We can easily fix this by adding a *lastUploadDate* property to the model. This property will tell us when the note was last uploaded:

```
Notes.NoteModel = function (config) {
    "use strict";
    this.id = config.id;
    this.dateCreated = config.dateCreated;
    this.title = config.title;
    this.narrative = config.narrative;
    this.lastUploadDate = config.lastUploadDate;
};
```

We will set this property to the correct value when we implement the *endNotesSync* function.

There is another piece of information that we need to know in addition to the last upload date. When was the last time the note was updated? If we don’t know this, we will not be able to determine if a version of a note uploaded to the server is newer than a version that already exists on the server.

To satisfy this requirement, we are going to add the *lastUpdateDate* property to the *NoteModel* class. The server will use the value of this property to update its notes cache:

```
Notes.NoteModel = function (config) {
    "use strict";
    this.id = config.id;
    this.dateCreated = config.dateCreated;
    this.title = config.title;
    this.narrative = config.narrative;
    this.lastUpdateDate = config.lastUpdateDate;
    this.lastUploadDate = config.lastUploadDate;
};
```

We need to set *lastUpdateDate* to the correct value when a note is updated. This will happen in the *controller*'s *onSaveNoteButtonTapped* function:

```
var onSaveNoteButtonTapped = function () {

    // Validate note.
    var titleEditor = $(noteTitleEditorSel);
    var narrativeEditor = $(noteNarrativeEditorSel);
    var tempNote = dataContext.createBlankNote();


    tempNote.title = titleEditor.val();
    tempNote.narrative = narrativeEditor.val();
    tempNote.lastUpdateDate = new Date();

    if (tempNote.isValid()) {

        if (null !== currentNote) {

            if (currentNote.title !== tempNote.title || currentNote.narrative !==
tempNote.narrative) {

                currentNote.title = tempNote.title;
                currentNote.narrative = tempNote.narrative;
                currentNote.lastUpdateDate = tempNote.lastUpdateDate; // Update last
upload date.
            }

        } else {

            currentNote = tempNote;
        }

        $.mobile.showPageLoadingMsg();

        dataContext.saveNote(currentNote);

        $.mobile.hidePageLoadingMsg();

        returnToNotesListPage();

    } else {
        $.mobile.changePage(invalidNoteDlgSel, defaultDlgTrsn);
    }
};
```

One last detail - if someone deletes a note on a device, how will the server know it needs to delete the note from its cache? We will resolve this by adding the *deleteAfterUpload* property to the *NoteModel* class, and modifying the delete note workflow.

The *NoteModel* class will look like this:

```
Notes.NoteModel = function (config) {
    "use strict";
    this.id = config.id;
    this.dateCreated = config.dateCreated;
    this.title = config.title;
    this.narrative = config.narrative;
    this.lastUpdateDate = config.lastUpdateDate;
    this.lastUploadDate = config.lastUploadDate || config.dateCreated;
    this.deleteAfterUpload = config.deleteAfterUpload || false;
};
```

Additionally, we will modify the controller's *onOKToDeleteNoteButtonTapped* function like so:

```
var onOKToDeleteNoteButtonTapped = function () {

    $.mobile.showPageLoadingMsg()

    //dataContext.deleteNote(currentNote);
    currentNote.deleteAfterUpload = true;
    dataContext.saveNote(currentNote);

    $.mobile.hidePageLoadingMsg();

    returnToNotesListPage();
};
```

Now we are not deleting the note immediately. We are simply flagging it for deletion after it is uploaded to the server. This gives the server the opportunity to remove the note from its cache.

We will also use the *deleteAfterUpload* property to hide the notes flagged for deletion so they are not rendered on the Notes List View. This requires a small change in the *controller*'s *renderNotesList* function, where we will inspect the flag, and add a list item for the note only if the flag has not been set to true:

```
var renderNotesList = function () {

    var notesList = dataContext.getNotesList();
    var view = $(notesListSelector);

    view.empty();

    if (notesList.length === 0) {
        $(noNotesCachedMsg).appendTo(view);
    } else {

        var liArray = [],
            notesCount,
            note,
            dateGroup,
            noteDate,
            i,
            ul,
            liHtml,
            liIcon,
            lastUpdateDate,
            lastUploadDate;

        notesCount = notesList.length,
        ul = $("<ul id=\"notes-list\" data-role=\"listview\"></ul>").appendTo(view);

        for (i = 0; i < notesCount; i += 1) {

            note = notesList[i];

            noteDate = (new Date(note.dateCreated)).toDateString();

            // We will not render notes that are marked for deletion.
            if (!note.deleteAfterUpload) {

                if (dateGroup !== noteDate) {
                    liArray.push("<li data-role=\"list-divider\">" + noteDate +
"</li>");
                    dateGroup = noteDate;
                }

                lastUpdateDate = new Date(note.lastUpdateDate).getTime();
                lastUploadDate = new Date(note.lastUploadDate).getTime();

                if (lastUpdateDate < lastUploadDate) {
                    liIcon = "<img src=\"img/loop_green.png\" alt=\"Uploaded\"
class=\"ui-li-icon\">";
                } else {
                    liIcon = "<img src=\"img/loop_red.png\" alt=\"Uploaded\"
class=\"ui-li-icon\">"; ;
                }

                liHtml = "<li>"
                + "<a data-url=\"index.html#note-editor-page?noteId=" + note.id + "\"
href=\"index.html#note-editor-page?noteId=" + note.id + "\">"
                + liIcon
                + "<div  class=\"list-item-title\">" + note.title + "</div>"
                + "<div class=\"list-item-narrative\">" + note.narrative + "</div>"
                + "</a>"
                + "</li>"

                liArray.push(liHtml);
            }

        }

        var listItems = liArray.join("");
        if (listItems.length !== 0) {
            $(listItems).appendTo(ul);
                ul.listview();
        } else {
            view.empty();
            $(noNotesCachedMsg).appendTo(view);
        }
    }
};
```

Another important change is that we are checking the length of the list in two places, and displaying a message when the list is empty. First, before iterating through the notes array:

```
$(noNotesCachedMsg).appendTo(view);
```

And then, after building the array of list items:

```
if (listItems.length !== 0) {
    $(listItems).appendTo(ul);
    ul.listview();
} else {
    view.empty();
    $(noNotesCachedMsg).appendTo(view);
}
```

This requires that we define the *noNotesCachedMsg* variable at the top of the controller:

```
var noNotesCachedMsg = "<div>Your notes list is empty.</div>";
```

With these modifications in place, we can switch our focus on the *endNotesSync* function. This function will process the information downloaded from the server.

Parsing Synchronization Results Sent By the Server

The notes synchronization is a two-way process. We are uploading the notes cached on the device, and receiving any updates stored on the server by other devices. The purpose of *endNotesSync* is to parse the data sent from the server and update the notes list on the device.

In the *dataContext* module, let's define *endNotesSync* as follows:

```
var endNotesSync = function (result, data, callback) {

    var note,
        syncResult,
        syncResults,
        idsOfNotesToDelete = [],
        notesListLength,
        i, j,
        okToContinue = true;

    if (result.success && data) {

        notesListLength = notesList.length;
        syncResults = data;

        // CRUD based on server results.
        for (i = 0; i < notesListLength; i += 1) {
            for (j = 0; j < syncResults.length; j += 1) {

                note = notesList[i];
                syncResult = syncResults[j];

                if (syncResult.isNewNote) {

                    // Add any new notes.
                    notesList.splice(0, 0, syncResult, note);
                } else {

                    // Update notes that made it to the server.
                    if (syncResult.note.id === note.id) {

                        note.title = syncResult.note.title;
                        note.narrative = syncResult.note.narrative;

                        note.lastUploadDate = new Date();

                        // Find out if we need to delete this note.
                        if (note.deleteAfterUpload) {
                            idsOfNotesToDelete.push(syncResult.note.id);
                        }
                    }
                }
            }
        }

        // Delete the notes that were marked for deletion and were successfully
uploaded.
        for (i = 0; i < idsOfNotesToDelete.length; i += 1) {

            notesListLength = notesList.length;
            okToContinue = true;

            for (j = 0; j < notesListLength && okToContinue; j += 1) {

                if (idsOfNotesToDelete[i] === notesList[j].id) {
                    notesList.splice(j, 1);
                    okToContinue = false;
                }
            }
        }

        saveNotesToLocalStorage();
    }
    callback();
};
```

We begin the function defining a few helper variables. The more interesting ones are *syncResults*, *syncResult* and *idsOfNotesToDelete*. We will use *syncResults* to store the payload sent by the server. As we will see later, this payload will consist of a JSON-serialized array of objects. Each of these objects will represent the result of the synchronization of a note, and it will have the following properties:

- *success*, which indicates whether the synchronization succeeded
- *note*, itself a JSON representation of the note in question
- *isNewNote*, which indicates if the note in question is a note that exists on the server, but isn't already cached on the client. This property tells us if the note was sent to the server by a different device.

The purpose of the *syncResult* variable is to serve as a placeholder to store one of the objects above.

The *idsOfNotesToDelete* array will store the ids of those notes that we need to delete. We will use it to loop through the notes cached on the device, and remove those whose ids exist in the array.

After defining our helper variables, we proceed to inspect the *result* and *data* arguments, as we are only interested in updating our notes cache if the AJAX operation succeeded. Then, we point the *syncResults* variable to the *data* argument, and proceed to update the notes cache.

This is a two-step process, where we first add new notes and update existing ones:

```
// Update based on server results.
for (i = 0; i < notesListLength; i += 1) {
    for (j = 0; j < syncResults.length; j += 1) {

        note = notesList[i];
        syncResult = syncResults[j];

        if (syncResult.isNewNote) {

            // Add any new notes.
            notesList.splice(0, 0, syncResult, note);
        } else {

            // Update notes that made it to the server.
            if (syncResult.note.id === note.id) {

                note.title = syncResult.note.title;
                note.narrative = syncResult.note.narrative;

                note.lastUploadDate = new Date();

                // Find out if we need to delete this note.
                if (note.deleteAfterUpload) {
                    idsOfNotesToDelete.push(syncResult.note.id);
                }
            }
        }
    }
}
```

Observe how we are iterating first over the array of cached notes, and then over the results sent by the server.

Adding new notes is easy, because the *synResult.isNewNote* tells us if the downloaded note is new. If the note is not new, we proceed to transfer its value to its equivalent on the client. This effectively updates the client's note with the values sent from the server.

We then set the *lastUploadDate* property of the note, and finally, perform a special step if the note was marked for deletion. The step consists of storing the *id* of the note marked for deletion in the *idsOfNotesToDelete* array. Remember that our Delete Note workflow consists of marking notes for deletion, and deleting the notes after they have been sent to the server. This allows us to inform the server, as well as other devices, that they also need to delete the note.

The second step in the processing of the results consists of iterating over the *idsOfNotesToDelete* array:

```
// Delete the notes that were marked for deletion and were successfully uploaded.
for (i = 0; i < idsOfNotesToDelete.length; i += 1) {

    notesListLength = notesList.length;
    okToContinue = true;

    for (j = 0; j < notesListLength && okToContinue; j += 1) {

        if (idsOfNotesToDelete[i] === notesList[j].id) {
            notesList.splice(j, 1);
            okToContinue = false;
        }
    }
}
```

This is where the notes marked for deletion are removed from the local cache. After this step, all we need to do is save the cache back to local storage:

```
saveNotesToLocalStorage();
```

Note that the last line of *endNotesSync* takes us back to the controller by invoking the callback function:

```
callback();
```

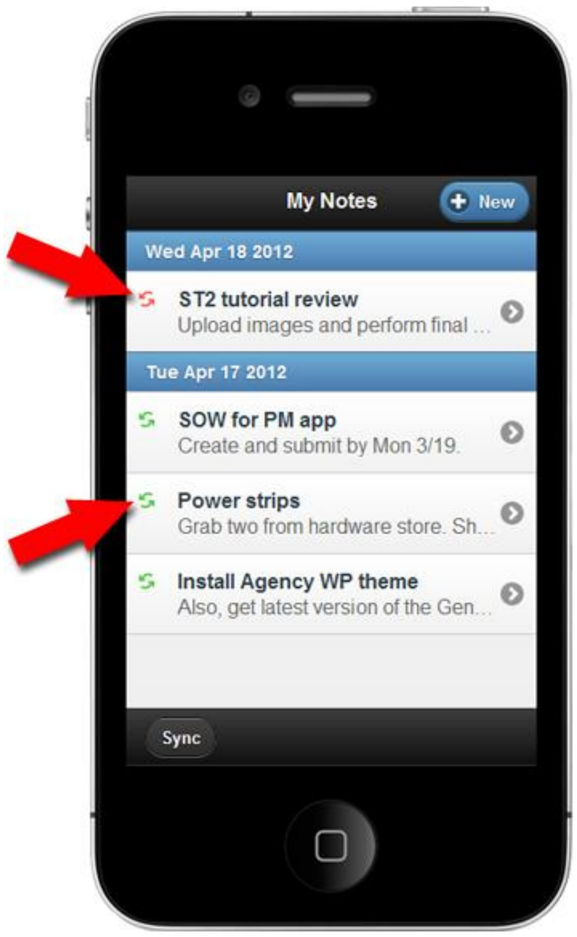
At this point we have a refreshed cache, and all the *controller* needs to do is re-render the notes list. This happens in the *controller* module, in the *onNotesSyncCompleted* function that we already created:

```
var onNotesSyncCompleted = function () {

    renderNotesList();
    $.mobile.hidePageLoadingMsg();
    $.mobile.loadingMessage = defaultLoadingMsg;

};
```

This function re-renders the notes list through a call to *renderNotesList*. Then, it hides the custom message we defined, and sets the Framework's *loadingMessage* back to its default value.

Using an Icon to Indicate a Note's Sync Status

We've just re-rendered the notes list after synchronization with the server took place. We have a refreshed notes cache, from which some notes may have been deleted, and some updated. In order to provide the user a form of visual feedback on the status of the notes cache, we are going to modify the Notes List page. Next to each note, we are going to render an icon that indicates the note's status:


My Notes
New
Wed Apr 18 2012
ST2 tutorial review
Upload images and perform final ...
Tue Apr 17 2012
SOW for PM app
Create and submit by Mon 3/19.
Power strips
Grab two from hardware store. Sh...
Install Agency WP theme
Also, get latest version of the Gen...
Sync

The red icon indicates that the note has not been synchronized, or that it was changed after it was last synchronized. The green icon indicates that the note has been synchronized with the server.

The perfect place to perform this modification is the *renderNotesList* function in the *controller* module. Let's modify this function as follows:

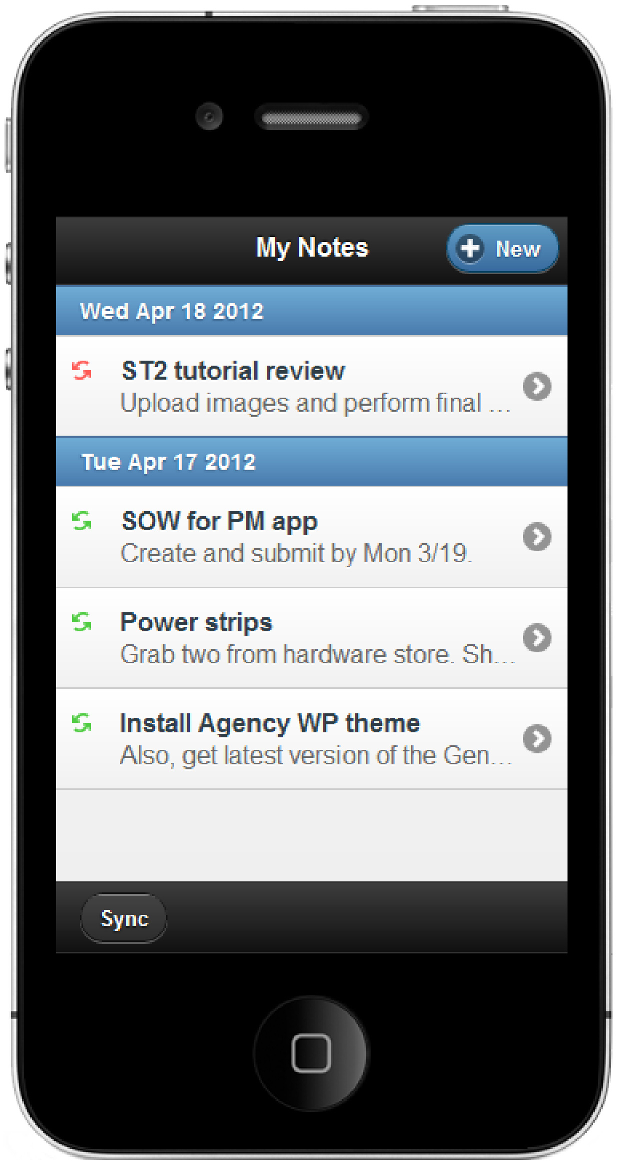
```
var renderNotesList = function () {

    var notesList = dataContext.getNotesList();
    var view = $(notesListSelector);

    view.empty();

    if (notesList.length === 0) {

        $(noNotesCachedMsg).appendTo(view);
    } else {

        var liArray = [],
            notesCount,
            note,
            dateGroup,
            noteDate,
            i,
            ul,
            liHtml,
            liIcon,
            lastUpdateDate,
            lastUploadDate;

        notesCount = notesList.length,
        ul = $("<ul id=\"notes-list\" data-role=\"listview\"></ul>").appendTo(view);

        for (i = 0; i < notesCount; i += 1) {

            note = notesList[i];

            noteDate = (new Date(note.dateCreated)).toDateString();

            // We will not render notes that are marked for deletion.
            if (!note.deleteAfterUpload) {

                if (dateGroup !== noteDate) {
                    liArray.push("<li data-role=\"list-divider\">" + noteDate +
"</li>");
                    dateGroup = noteDate;
                }

                lastUpdateDate = new Date(note.lastUpdateDate).getTime();
                lastUploadDate = new Date(note.lastUploadDate).getTime();

                if (lastUpdateDate < lastUploadDate) {
                    liIcon = "<img src=\"img/loop_green.png\" alt=\"Uploaded\"
class=\"ui-li-icon\">";
                } else {
                    liIcon = "<img src=\"img/loop_red.png\" alt=\"Uploaded\"
class=\"ui-li-icon\">"; ;
                }

                liHtml = "<li>"
                + "<a data-url=\"index.html#note-editor-page?noteId=" + note.id + "\"
href=\"index.html#note-editor-page?noteId=" + note.id + "\">"
                + liIcon
                + "<div  class=\"list-item-title\">" + note.title + "</div>"
                + "<div class=\"list-item-narrative\">" + note.narrative + "</div>"
                + "</a>"
                + "</li>"

                liArray.push(liHtml);
            }

        }

        var listItems = liArray.join("");
        if (listItems.length !== 0) {
            $(listItems).appendTo(ul);
            ul.listview();
        } else {
            view.empty();
           $(noNotesCachedMsg).appendTo(view);
        }
    }
};
```

Besides the three new variables - *liIcon*, *lastUpdateDate* and *lastUploadDate* - the lines we need to pay attention to are the following:

```
// We will not render notes that are marked for deletion.
if (!note.deleteAfterUpload) {

    if (dateGroup !== noteDate) {
        liArray.push("<li data-role=\"list-divider\">" + noteDate + "</li>");
        dateGroup = noteDate;
    }

    lastUpdateDate = new Date(note.lastUpdateDate).getTime();
    lastUploadDate = new Date(note.lastUploadDate).getTime();

    if (lastUpdateDate < lastUploadDate) {
        liIcon = "<img src=\"img/loop_green.png\" alt=\"Uploaded\" class=\"ui-li-
icon\">";
    } else {
        liIcon = "<img src=\"img/loop_red.png\" alt=\"Uploaded\" class=\"ui-li-
icon\">"; ;
    }

    liHtml = "<li>"
    + "<a data-url=\"index.html#note-editor-page?noteId=" + note.id + "\"
href=\"index.html#note-editor-page?noteId=" + note.id + "\">"
    + liIcon
    + "<div  class=\"list-item-title\">" + note.title + "</div>"
    + "<div class=\"list-item-narrative\">" + note.narrative + "</div>"
    + "</a>"
    + "</li>"

    liArray.push(liHtml);
}
```

After checking the *deleteAfterDownload* property of the note, and creating the list header with the note's date, we compare the last update date of the note to its last upload date. Depending on the results of the comparison, we load the *liIcon* variable with an image template for the red or green icon. Then, we add the icon to the list item's template. Evidently, we will store the images in the *NotesApp/img* directory, which we need to create.

The result of the modification is, of course, that notes that have never been uploaded, or have been updated after being uploaded, will render with the red icon. Notes that have been uploaded after being changed, will render with the green icon.


My Notes
New
Wed Apr 18 2012
ST2 tutorial review
Upload images and perform final ...
Tue Apr 17 2012
SOW for PM app
Create and submit by Mon 3/19.
Power strips
Grab two from hardware store. Sh...
Install Agency WP theme
Also, get latest version of the Gen...
Sync

This concludes our workflow on the client side of the application. Now we can focus on the server side.

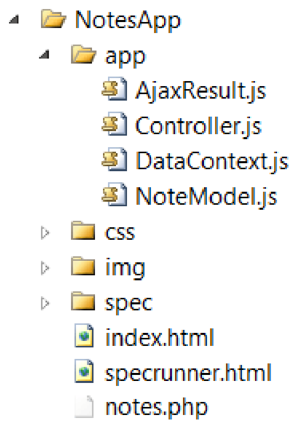
As a detailed discussion of server-side code is out of the scope of this book, we are going to create a very simple server-side handler that will receive the notes sent by the application, and will send updated notes back to it.

In order to simulate what happens in a real-world application - for example, a database access operation - we will introduce an artificial delay between the time the server receives the notes and the time it sends the updated notes back to our jQuery Mobile page.

We are going to create two versions of this server-side handler. One using PHP, and one using C#.

The Server-Side Code, PHP Version

We will call our handler notes.php:



The first thing we are going to do in it is create a class to represent a note:

```
<?php

// Class to represent a note.
class Note {
      function __construct($id, $dateCreated, $title, $narrative, $lastUpdateDate) {
            $this->id = $id;
            $this->dateCreated = $dateCreated;
            $this->title = $title;
            $this->narrative = $narrative;
        $this->lastUpdateDate = $lastUpdateDate;
      }
}
```

The *Note* class on the server is the equivalent of the *NoteModel* class on the client. We will see how the server-side code will use instances of this class in a minute.

Next, we are going to create a class to represent the result of a note synchronization:

```
// Class to represent sync result.
class NoteSyncResult {
	function __construct($success, $isNewNote, $note) {
		$this->success = $success;		// Used to let client know the 
note it sent was synchronized.
		$this->isNewNote = $isNewNote;		// Used to let client know this 
a note created by another client.
		$this->note = $note;
	}
}
```

Our server handler will send a JSON-serialized instance of this class to the client for each note that exists on the server. The *success* property tells the client application whether the note was successfully synched on the server. The *isNewNote* property indicates whether the note is new, which means that it was created by another client application and uploaded to the server. The *note* property is a serialized instance of the note. This note may be new, or may already exist on the client, but has been updated on the server.

Finally, let's see how the handler processes the notes uploaded by the application, and uses these two classes to build a response and send it back to the application.

First, we will define a couple of variables that will hold the uploaded notes and the payload that we will send back to the application:

```
$notes = json_decode($_POST["notes"]);
$syncResults = array();
```

Notice how we need to decode the posted notes, as they are JSON-formatted.

Then, we are going to iterate through the uploaded notes. For each note, we create a *NoteSyncResult* instance that will represent the result of our simulated synchronization:

```
foreach($notes as $key => $note) {

	$serverNote = new Note($note->id, $note->dateCreated, $note->title, $note-
>narrative, $note->lastUpdateDate);

	// This note would be added to the server's database here. (We're not 
discussing this subject in the book.)

	$syncResult = new NoteSyncResult(true, false, $serverNote);
	array_push($syncResults,$syncResult);
}

// New notes would be added to results here. (We're not discussing this subject in 
the book.)
```

This is a simulated synchronization because we are not saving the uploaded notes, or comparing them to an existing list of notes that we keep in a database. Although it's out of the scope of this book, it is not hard to see how this scenario could be implemented within a loop similar to the one above.

In the loop, we create an instance of the *Note* class for each uploaded note, and an instance of the *NoteSyncResult* class, which indicates that the synchronization was successful, and contains the *Note* instance. We store the *NoteSyncResult* instance in the *syncResults* array.

Next, we serialize the *syncResults* array, and send it back to the client application:

```
// Artificial delay to simulate long-running op on the server.
sleep(3);

header('Cache-Control: no-cache, must-revalidate');
header("content-type:application/json");
echo(json_encode($syncResults));
```

Notice how we are introducing a delay of 3 seconds to simulate how a real-world server would behave, where we would need to compare the uploaded notes to a list stored in a database, and perform additions, updates or deletions in the database.

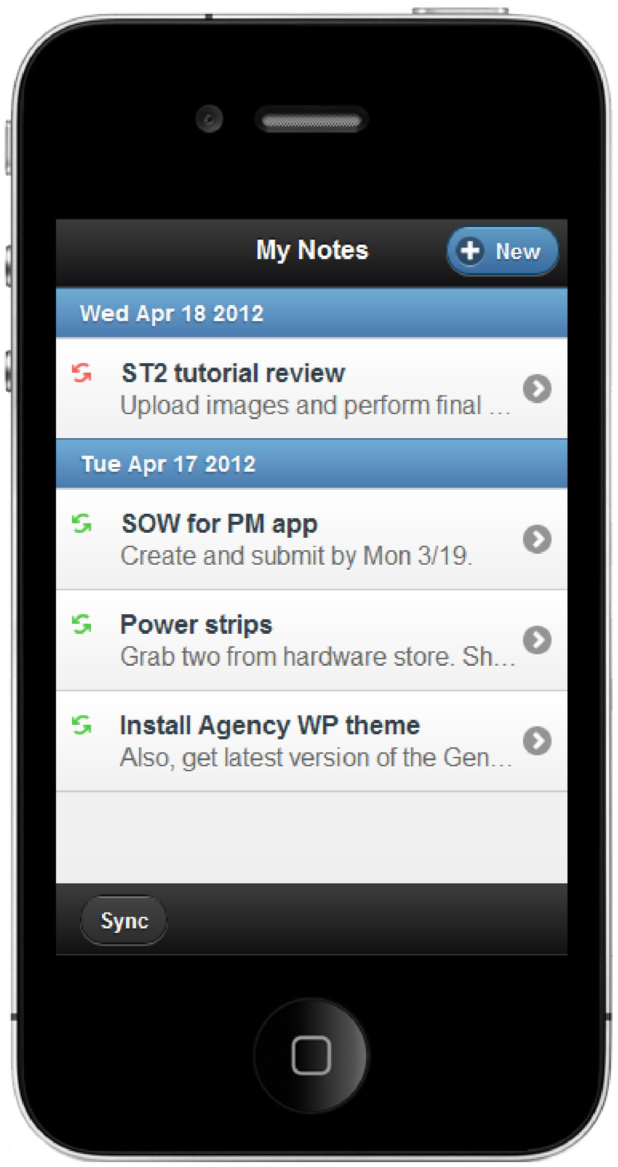
This is all it takes to create the PHP version of the server-side code against which we can test our application. Remember that, in the application, we set the URL of the server handler through the *appServerUrl* variable in the *controller* module. The value of the variable should be *notes.php*.

Here's the entire handler:

```
<?php

// Class to represent sync result.
class Note {
	function __construct($id, $dateCreated, $title, $narrative, $lastUpdateDate) {
		$this->id = $id;
		$this->dateCreated = $dateCreated;
		$this->title = $title;
		$this->narrative = $narrative;
	  $this->lastUpdateDate = $lastUpdateDate;
	}
}

// Class to represent sync result.
class NoteSyncResult {
	function __construct($success, $isNewNote, $note) {
		$this->success = $success;		// Use to let client know the
note it sent was synchronized.
```

```
		$this->isNewNote = $isNewNote;		// Used to let client know this a note created by another client.
		$this->note = $note;
	}
}


$notes = json_decode($_POST["notes"]);
$syncResults = array();

foreach($notes as $key => $note) {

	$serverNote = new Note($note->id, $note->dateCreated, $note->title, $note->narrative, $note->lastUpdateDate);

	// This note would be added to the server's database here. (We're not discussing this subject in the book.)

	$syncResult = new NoteSyncResult(true, false, $serverNote);
	array_push($syncResults,$syncResult);
}

// New notes would be added to results here. (We're not discussing this subject in the book.)

// Artificial delay to simulate long-running op on the server.
sleep(3);

header('Cache-Control: no-cache, must-revalidate');
header("content-type:application/json");
echo(json_encode($syncResults));
```

If we run the application and upload some notes, we should see the icons in the Notes List page switch from red to green:

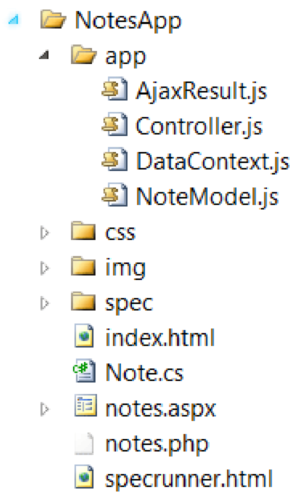

My Notes
New
Wed Apr 18 2012
ST2 tutorial review
Upload images and perform final …
Tue Apr 17 2012
SOW for PM app
Create and submit by Mon 3/19.
Power strips
Grab two from hardware store. Sh…
Install Agency WP theme
Also, get latest version of the Gen…
Sync

We can also change the *success* or *isNewNote* properties of some of the *NoteSyncResult* class instances we create in the PHP page. This allows us to simulate synchronization failures, or sending new notes to the application.

If we simulate synchronization failures, we should see the icons remain red. If we send new notes to the application, the notes should render in the Notes List page, and remain cached on the device.

The Server-Side Code, C# Version

The C# version of the server-side code that will handle notes synchronization is very similar in features to the PHP version. Let's create the notes.aspx file that will serve as the server-side handler, and the Note.cs file, which we will use to store a couple of helper classes:



In the notes.aspx.cs file, let's add the following code:

```
namespace NotesApp
{
    public partial class notes : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            var notesJson = Request.Form["notes"];
            if (null != notesJson)
            {
                ServerNote[] notes =
JsonConvert.DeserializeObject<ServerNote[]>(notesJson) ;

                List<NoteSyncResult> results = new List<NoteSyncResult>();

                foreach (var note in notes)
                {

                    ClientNote clientNote = new ClientNote()
                    {
                        id = note.Id,
                        dateCreated = note.DateCreated,
                        lastUpdateDate = note.LastUpdateDate,
                        lastUploadDate = DateTime.Now,
                        title = note.Title,
                        narrative = note.Narrative
                    };
                    // This note would be added to the server's database here. (We're
not discussing this subject in the book.)

                    // We are going to assume that everything went well and send each
note back to the client.
                    NoteSyncResult result = new NoteSyncResult()
                    {
                        success = true,
                        note = clientNote,
                        isNewNote = false
                    };

                    results.Add(result);
                }

                // New notes would be added to results here. (We're not discussing
this subject in the book.)

                Thread.Sleep(3000); // Artificial delay to simulate long-running op
on the server.


                var serializedResults = JsonConvert.SerializeObject(results);
                Response.ContentType = "application/json";
                Response.Write(serializedResults);
                Response.End();
            }

        }
    }
}
```

The processing of the uploaded notes takes place in the Page_Load method, where we first capture the uploaded notes and load them into an array of *ServerNote* instances. Observe that we are using the JSON.NET library to deserialize the JSON-encoded list of notes uploaded by the application.:

```
var notesJson = Request.Form["notes"];
if (null != notesJson)
{
    ServerNote[] notes = JsonConvert.DeserializeObject<ServerNote[]>(notesJson) ;
```

The *ServerNote* class is a server-side representation of a note. We will use instances of this class to perform the synchronization of the notes sent by the application to the list stored on the server. *ServerNote* resides in the Note.cs file, and this is its definition:

```
public class ServerNote
{
    public string Id { get; set; }
    public string Title { get; set; }
    public string Narrative { get; set; }
    public DateTime DateCreated { get; set; }
    public DateTime LastUpdateDate { get; set; }
    public DateTime LastUploadDate { get; set; }
}
```

After deserializing the uploaded notes, we define a list of *NoteSyncResult* instances, which we will use to send the results of the synchronization back to the application:

```
List<NoteSyncResult> results = new List<NoteSyncResult>();
```

The *NoteSyncResult* class represents the result of a note synchronization. We will place it in the Note.cs file as well:

```
public class NoteSyncResult
{
    public bool success {get;set;}
    public bool isNewNote { get; set; }
    public ClientNote note { get; set; }
}
```

The *success* property tells the client application whether the note was successfully synched on the server. The *isNewNote* property indicates whether the note is new, which means that it was created by another client application and uploaded to the server. The *note* property is a serialized instance of the note. This note may be new, or may already exist on the client.

Back in the *Page_Load* function, we proceed to iterate over the array of notes sent by the application:

```
foreach (var note in notes)
{

    ClientNote clientNote = new ClientNote()
    {
        id = note.Id,
        dateCreated = note.DateCreated,
        lastUpdateDate = note.LastUpdateDate,
        lastUploadDate = DateTime.Now,
        title = note.Title,
        narrative = note.Narrative
    };

    // This note would be added to the server's database here. (We're not discussing
this subject in the book.)

    // We are going to assume that everything went well, and send each note back to
the client.
    NoteSyncResult result = new NoteSyncResult()
    {
        success = true,
        note = clientNote,
        isNewNote = false
    };

    results.Add(result);
}

// New notes would be added to results here. (We're not discussing this subject in
the book.)
```

We will use the ClientNote class, which also resides in the Note.cs file, to send JSON-serialized data to the application:

```
public class ClientNote
{
    public string id { get; set; }
    public string title { get; set; }
    public string narrative { get; set; }
    public DateTime dateCreated { get; set; }
    public DateTime lastUploadDate { get; set; }
    public DateTime lastUpdateDate { get; set; }
}
```

Back in the notes.aspx.cs page, in the loop, we create an instance of the *ClientNote* class for each uploaded note, and an instance of the *NoteSyncResult* class, indicating that the synchronization was successful. Each *NoteSyncResult* instance contains a *Note* class instance. After we finish processing each note, we serialize the *syncResults* array and send it back to the client application:

```
Thread.Sleep(3000); // Artificial delay to simulate long-running op on the server.
var serializedResults = JsonConvert.SerializeObject(results);
Response.ContentType = "application/json";
Response.Write(serializedResults);
Response.End();
```

Notice how we are introducing a 3-second delay to simulate how a real-world server would behave, where we would need to compare the uploaded notes to a list stored in a database, and perform additions, updates or deletions in the database.

This is all it takes to create the C# version of the server side code against which we can test our application. Remember that, in the application, we set the URL of the server handler through the *appServerUrl* variable in the *controller* module. The value of the variable should be *notes.aspx*.

Where Are We?

In this chapter we modified our application so it is able to keep the notes saved on the device synchronized with a server.

This process allowed us to learn how to add time stamps to the data model of a note, how to create and handle the AJAX requests that allow communications with a server, and how to modify the visual style of the items in a jQuery Mobile list view.

We Made It!

I hope this has been a fruitful journey for you. While creating the Notes Application, we became familiar with the building blocks of a jQuery Mobile framework, and learned a number of important practices such as how to craft beautiful user interfaces, create and style list views, capture input using form elements, save data on the device, and synchronize data with a server.

This knowledge will definitely help you take the next steps in your path to becoming a great mobile applications developer.

Keep in Touch

Your feedback is very important to me. Please send me your comments or questions through my blog at http://miamicoder.com.