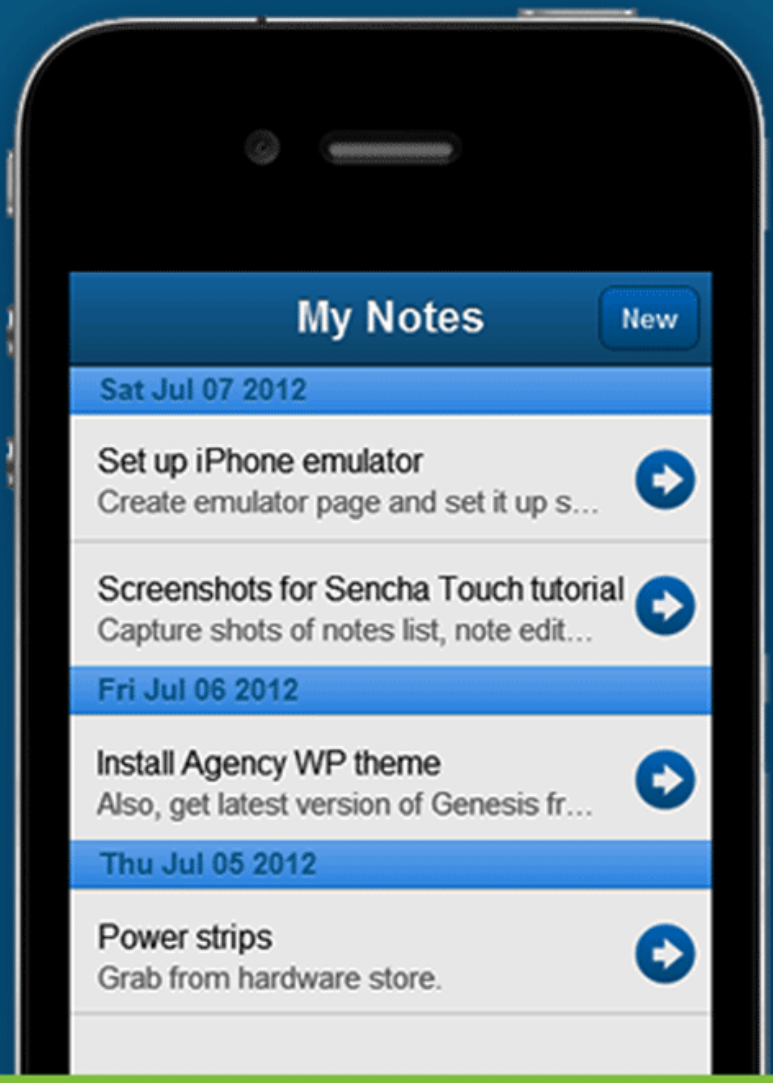
Building a Sencha Touch Application

A Hands-On Introduction to Sencha Touch for Mobile Application Developers



Jorge Ramon

miamicoder.com

Building a Sencha Touch Application

A Hands-On Tutorial for Mobile Application Developers

Book Version: 2.00

www.miamicoder.com

Chapter 2: Rendering Cached Notes

What we will learn in this chapter

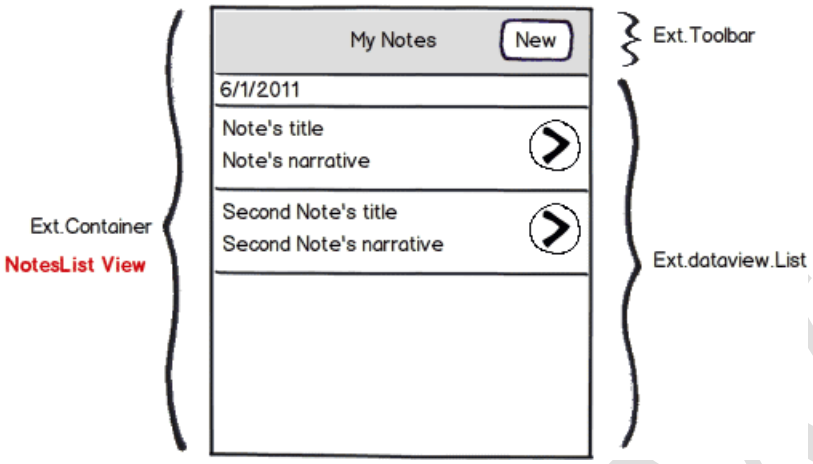
In this chapter we are going to start building the NotesList view. This view is an example of a list-based user interface, one of the most popular user interface styles in mobile applications.

In the process of building the NotesList view, we will learn the following topics:

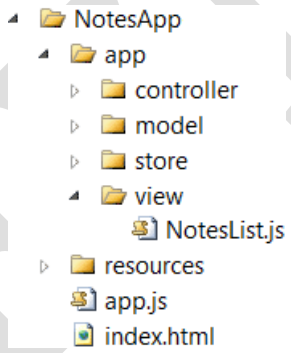
- How to create a list-based user interface using Sencha Touch's List component.
- How to create a Sencha Touch model, and how to define model validations.
- How to create a data store in Sencha Touch, and how a store and its model work together.
- How to bind a Sencha Touch List to a data store.
- How to modify the look of the items of a Sencha Touch List.
- How to use the Sencha Touch TitleBar component.

Defining a view in Sencha Touch

We already know that the main view of our application will render a list of notes. To build this view, we will use an instance of the *Container* class, which will host a *Toolbar* and a *List* component:



As a first step, we will create the NotesList.js file in the *view* directory:



In the NotesList.js file, we will then define the NotesList class like so:

```
Ext.define('NotesApp.view.NotesList', {
    extend: 'Ext.Container',
    requires:['Ext.TitleBar'],
    alias: 'widget.noteslistview',
    config: {
        layout: {
            type: 'fit'
        },
        items: [{
            xtype: 'titlebar',
```

```
            title: 'My Notes',
            docked: 'top',
            items: [
                {
                    xtype: 'button',
                    text: 'New',
                    ui: 'action',
                    itemId: 'newButton',
                    align: 'right'
                }
            ]
        }]
    }
});
```

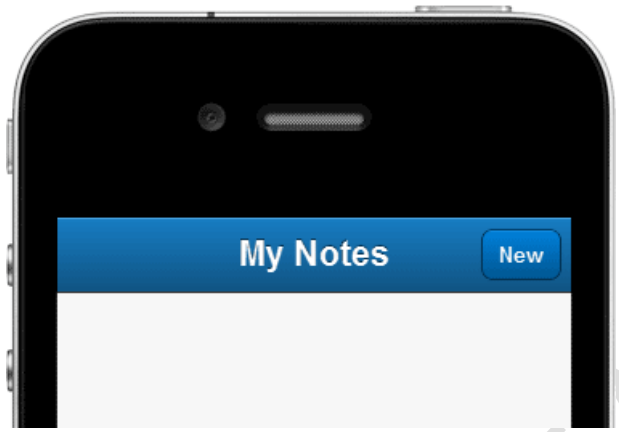
We are using *Ext.define* and the *extend* config to define an extension to the *Ext.Container* class.

As we will use a *TitleBar* instance for the view's toolbar, we include the *requires* config, which guarantees that the *Ext.Loader* will download the source of the *TitleBar* component.

The *Ext.Loader* singleton is very helpful during application development, as it allows applications to download their dependencies on the fly.

In the *TitleBar* configuration, the *docked* config allows us to dock the *TitleBar* to the top region of the view. The *TitleBar* will in turn contain one *Button*, the New button, which will allow our users to activate the NoteEditor view when they need to create a new note. Additionally, we use the *align* config to place the New button on the right end of the *TitleBar*.

Setting the value of the *ui* config to *action* gives the New button a distinctive look, indicating that it represents the default button on the view.



The *alias* config of the NotesList class will allow us to refer to the class by its *xtype*, which is simply a shortcut for its full name, as we will see next.

Adding the view to the application

With the *NotesList* class defined, it is time to make the *Application* aware of it. Back in the app.js file, we are going to add the *views* config to the *application* method:

```
Ext.application({
    name: 'NotesApp',

    views: ['NotesList'],

    launch: function () {

        var notesListView = {
            xtype: 'noteslistview'
        };

        Ext.Viewport.add([notesListView]);

    }
});
```


We inform the *Application* that it has a dependency on the NoteList view by using the *views* config:

```
views: ['NotesList'],
```

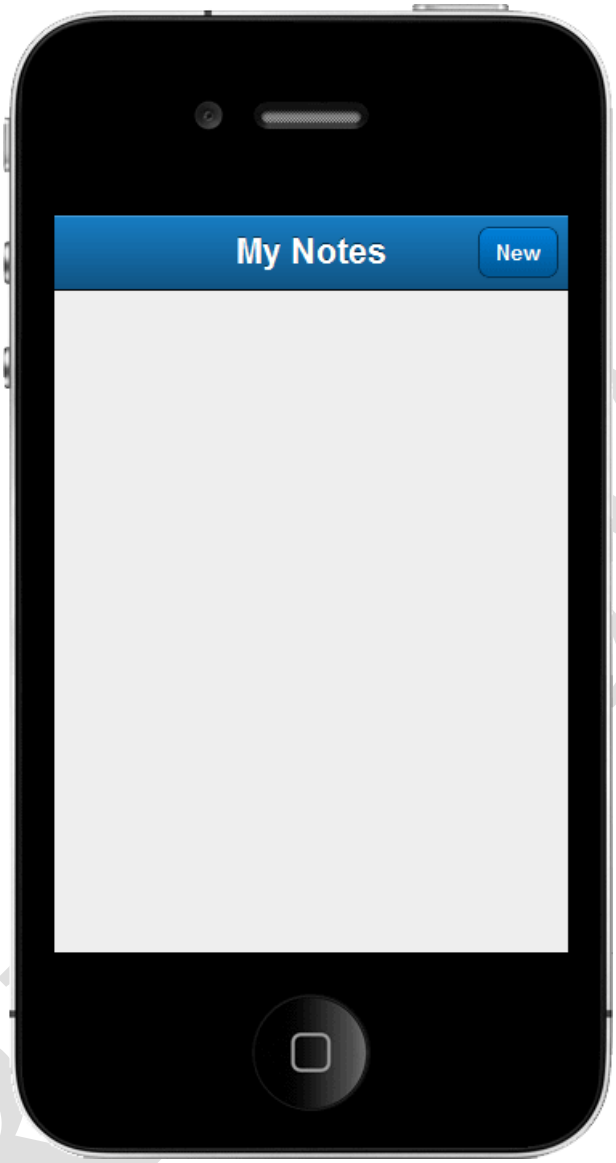
By default, the *Application* class expects its models, views, controllers, stores and profiles to exist in the app/model, app/view, app/controller, app/store, and app/profile directories. If we follow this convention, we can define models, views, controllers, stores and profiles using the last part of their names, as we do here. If we use a different directory structure, we can still refer to the application's components using their fully qualified names.

After adding the NotesList view to the *views* config, we create an instance of this view in the app's *launch* method. Finally, we add the view to the app's viewport:

```
var notesListView = {
    xtype: 'noteslistview'
};

Ext.Viewport.add([notesListView]);
```

It is time to check out how the NotesList view looks. Opening the index.html file in our favorite WebKit-powered browser should produce something similar to the screenshot below:



Configuring the notes list

Our next step in the NotesList view is to add the component that will render the list of cached notes. Back in the NotesList.js file, we will add an *Ext.dataview.List* instance to the *items* config like so:

```
Ext.define('NotesApp.view.NotesList', {
    extend: 'Ext.Container',
    requires:['Ext.TitleBar','Ext.dataview.List'],
    alias: 'widget.noteslistview',
```



```
    config: {
        layout: {
            type: 'fit'
        },
        items: [{
            xtype: 'titlebar',
            title: 'My Notes',
            docked: 'top',
            items: [
                {
                    xtype: 'button',
                    text: 'New',
                    ui: 'action',
                    itemId: 'newButton',
                    align: 'right'
                }
            ]
        },{
              xtype: 'list',
              store:[],
              itemId: 'notesList',
              loadingText: 'Loading Notes...',
              emptyText: '<div>No notes found.</div>',
              onItemDisclosure: true,
              itemTpl: '<div>{title}</div><div>{narrative}</div>'
        }]
    }
});
```

Notice that we added the *Ext.dataview.List* entry to the *requires* config of the view.

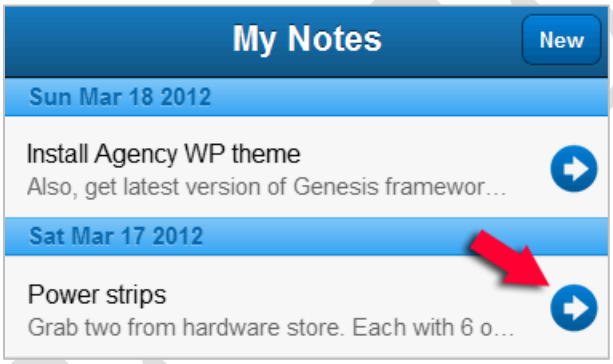
Now, let's look at the list's definition in detail:

```
{
    xtype: 'list',
    store:[],
    itemId: 'notesList',
    loadingText: 'Loading Notes...',
    emptyText: '<div>No notes found.</div>',
    onItemDisclosure: true,
    itemTpl: '<div>{title}</div><div>{narrative}</div>'
}
```


As we did with the *TitleBar* instance, we are using the *xtype* config to lazy-instantiate the list. Although we are using an empty array as the list's store, in the next section of this chapter we will create the *Store* component that will feed the list.

The *emptyText* and *itemTpl* configs allow us to control how the "empty list" message and the list's items will render.

Setting the *onItemDisclosure* config to *true* causes the list to render a *disclose* icon next to each list item:



Later in the book we will use the *disclose* icon as well as the *itemId* config to define a handler method for the list's *disclose* event. This is how we will load an existing note into the NoteEditor view.