Sencha Touch IN ACTION

Jesus Garcia
Anthony De Moss
Mitchell Simoens

MANNING

MEAP Edition
Manning Early Access Program
Sencha Touch in Action version 8

For more information on this and other Manning titles go to
www.manning.com

brief contents

1

Introduction to Sencha Touch

This chapter covers

- The problems Sencha Touch aims to solve
- The Sencha Touch UI palette
- Thinking like a mobile web developer

You're on the hook to build a mobile application. Perhaps you've been tasked in a project or have a great idea and want to make it become a reality. Either way, to build this application, you're going to at least have to learn Objective C for iOS or Java for Android. It should be no surprise that if you're to support both types of devices, you'll have to learn and master both languages, unless you choose a third-party native framework like Sencha Touch to bridge the gap between the devices.

Chances are you have experience in HTML, CSS and JavaScript and want to leverage what you know to build your mobile application. Being able to tie in your prior experience is part of what makes Sencha Touch a good choice for folks like you and me, since it offers a wide range of UI widgets to choose from, as well as robust data, layout, and component models.

In this chapter, we'll begin our journey into the world of Sencha Touch, where we'll discuss what Sencha Touch is and the problems it aims at solving, such as being able to develop cross-platform user interfaces with HTML5. We'll then look at the widgets that the framework provides. Lastly, we'll discuss some of the ways you should think about developing your mobile application, in hopes to avoid future performance issues.

What we'll find along the way is that developing mobile applications with this framework is not as difficult as with other technologies, such as Objective C or Java.

1.1 What is Sencha Touch?

Sencha Touch was born out of many of the ideas and culture from the venerable Ext JS framework and is the first mobile HTML5 JavaScript framework, such as the Sencha class system.

Sencha Touch solves the cross-platform mobile app app development problem of giving developers the tools necessary to build cross-platform applications that mimic natively compiled applications while making full use of HTML5 and CSS3. It also allows developers who have years of experience on the web develop cross-platform mobile apps that can exist solely in the web or be deployed in an app store with either the Sencha native packager or tools like phonegap.

At the time of this writing, Sencha Touch runs on mobile webkit-based browsers in iOS (iPhone, iPad) devices, as well as Android phones and tablets.

READ ABOUT HTML5

HTML5 is a collection of technologies that include enhancements to HTML itself, CSS3 and even JavaScript. It is changing the way we develop web applications by providing JavaScript API access to do things like talk directly to a graphics card (WebGL), Sound manipulation and even offline storage. While it's not completely necessary to know everything about HTML5 to use Sencha Touch, it is a good idea to get the basics down. A great site to learn about HTML5 is http://www.html5rocks.com/.

An excellent example of a Sencha Touch application is Checkout, by Steffen Hiller, illustrated below running on an iPad. Here, you can see an application that makes use of Sencha Touch providing a rich UI with HTML5.

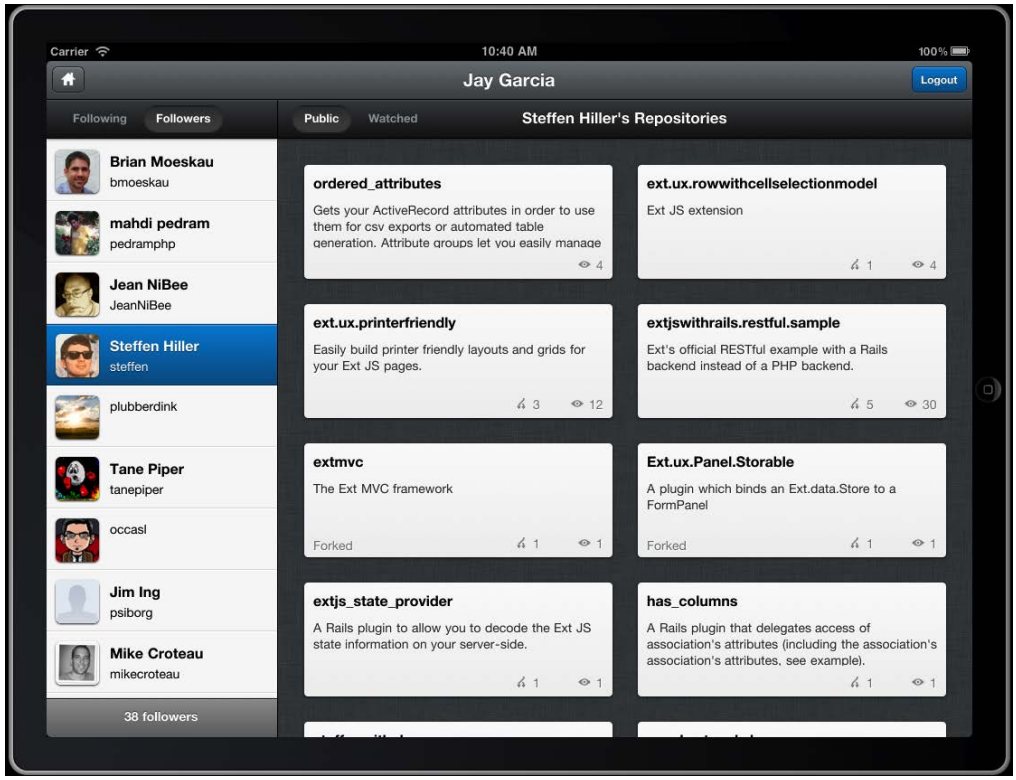

Figure 1.1 A full-screen Sencha Touch application, known as Checkout, which allows you to keep tabs on your Github account and followers. You can learn more about it via http://checkout.github.com.

To see other Sencha Touch applications, you can point your browser to http://sencha.com/apps and view the Sencha Touch app gallery. Here you can preview them via images, and even see them work live via embedded links.

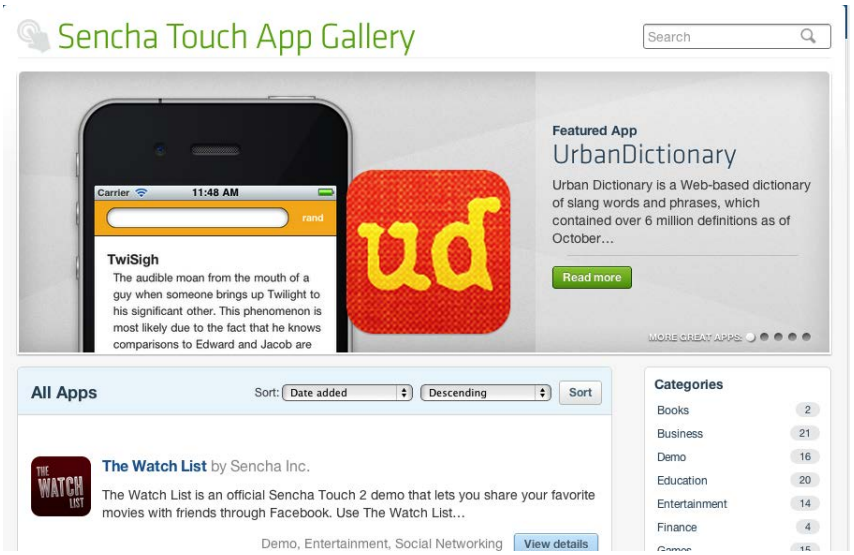

Figure 1.2 The Sencha Touch app Gallery

Much like Ext JS, Sencha Touch brings the native application feel by means of a clever blend of HTML5, CSS3 and JavaScript, all optimized for the best mobile experience possible given the constraints of mobile devices today, such as limited CPU and memory for our applications.

Also like its big brother, Sencha Touch is designed to be extensible and modifiable out of the proverbial box.

1.1.1 What Sencha Touch is not

While Sencha Touch works on desktop WebKit browsers, like Safari and Chrome (to a limited capacity), it is not designed for desktop Rich Internet Applications. Upon its release, lots of developers balked at the idea of this framework not functioning in Firefox or Internet Explorer.

The fact is that Sencha Touch is aimed at the development of mobile applications only. This means that if you've only developed applications with Firefox, IE and their respective debugging toolkits, you're going to have to leave your comfort zone.

"Sencha Touch" !== "Ext JS"

If you're a veteran Ext JS developer, you will feel right at home when learning Sencha Touch. It is important to know that there are some significant differences between the two libraries. Throughout this book, we'll try to point out some of the differences, but

cannot cover every possible point. If you have doubts, always check with the API documentation.

If you're unfamiliar with Ext JS and need to develop applications for the desktop web, check out Ext JS in Action.

1.1.2 Lots of wiring under the hood

To make user of mobile interactions, Sencha Touch comes with a gesture library, allowing you to easily hook into gesture-based events, such as tap, pinch and swipe. One way Sencha Touch to comes close to the feel of native applications is by means of a custom physics-based Scroller class, which uses hardware-accelerated CSS3 transitions and includes key variables like slide friction and spring effects.

1.1.3 Hardware compatibility

Many mobile touch-screen smart devices are entering the marketplace today, which is driving the increase in demand for mobile applications. While Sencha Touch aims at 100% compatibility across all mobile devices, the best user experience is on iOS and high-powered Android devices.

Why the difference in user experience?

The main reason for the difference in user experience between iOS and Android has to do with the physical computing power of each device and how the each individual device manufacturer compiles mobile WebKit for their device. Apple devices include a GPU, and compiles mobile Safari with GPU acceleration enabled. Most Android devices do not have dedicated GPUs. And for the ones who do, manufacturers typically do not compile mobile WebKit to enable GPU acceleration.

Sencha Touch applications do such a great job at mimicking how native applications look and feel that it's often easy to get lost in the fact that you're using a web-based application. This especially holds true when the mobile WebKit toolbars are hidden from view.

1.1.4 Full-screen goodness

Figure 1.2 illustrates how a Sencha Touch application looks when accessing the application via mobile Safari (left), versus accessing the application via a shortcut on your home screen.

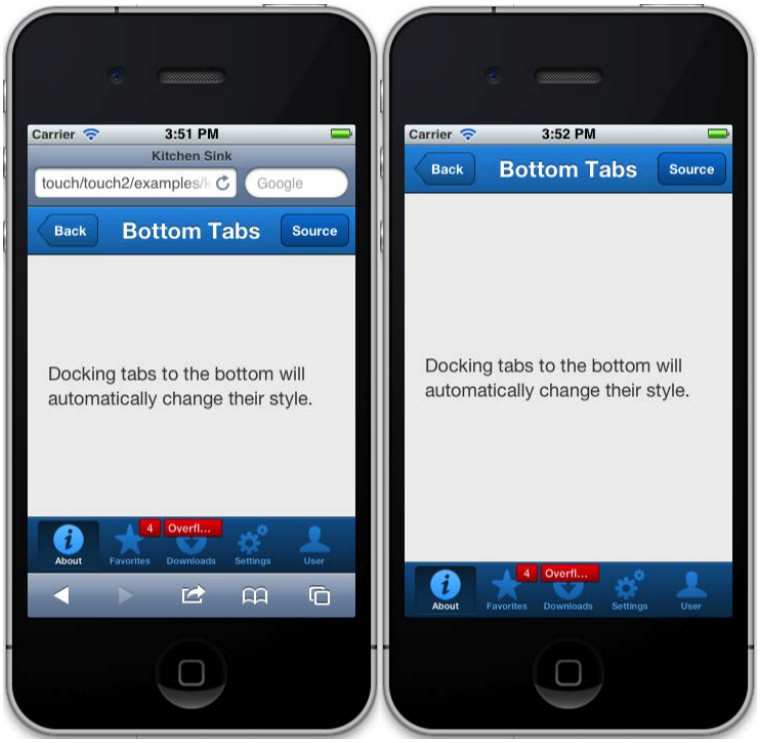

Figure 1.3 Looking at the Sench Touch kitchen sink example via Mobile Safari directly (left), or a shortcut on your home screen (right).

After looking at figure 1.3, it should be clear that a full-screen view of a Sencha Touch application looks very close to a native application. Also having your app in full screen means that much needed screen real estate is usable in your applications.

Sencha Touch offers a lot when it comes to UI widgets, but it's certainly just the surface of this framework. If you're like me, you probably want to just skip ahead and dive into code. But before we get our hands dirty, we should browse through the library and discuss some of its features.

1.2 A 10,000 foot view

If you have glanced at the Sencha Touch API documentation, the sheer number of classes in the library might have overwhelmed you. To make sense of it all, we must understand that these classes can be broken down into a few major groups.

Table 1.1 below describes the major groups that Sencha Touch is broken down in.

Table 1.1 Describing the various sections of purpose in the Sencha Touch framework.

Group	Purpose
Platform	The shared base of Sencha Touch and Ext JS v4 and is the bulk of the code for Sencha Touch.
Layout	A set of managers for visually organizing widgets on screen.
Utilities	Utilities is a group of useful odds and ends for the framework.
Data	Data is the information backbone for Sencha Touch and includes the means for retrieving, reading and storing data.
Style	Sencha Touch's theme is automatically generated via Sass (Syntactically Awesome Style Sheets).
MVC	Sencha Touch comes with an MVC framework for your application.
UI Widgets	A collection of visual components that your users will interact with.

The base library for Sencha Touch is known as Sencha Platform. Sencha Platform is based off of Ext JS 4.0 but is much improved in many respects. For example the class system in Sencha Touch resembles that of Ext JS 4, however it is much more advanced in many ways, such as a feature known as the Config System, which allows Sencha Touch to work much more effectively than Ext JS.

The Layout portion of Sencha touch is implemented by some of the UI widgets and is the code responsible for visually organizing items on the screen. The layouts are responsible for implementing transitional animations if configured to do so.

The Utilities section of the framework is a collection of useful bits of functionality that are often implemented by the framework and can be implemented by you. For instance, the List widget implements XTemplate to paint HTML fragments on screen. However, the XTemplate is open for you to use to do the same in your own custom widget.

The Data package is a group of classes that gives Sencha Touch the ability to fetch and read data from a myriad of sources, including mobile WebKit's HTML5 Session, Local and Database Storage methods. Sencha Touch can read data in a variety of formats including XML, Array, JSON, and Tree (nested).

The style area of the framework is not something that you typically deal with on a day-to-day basis, but is something that is worth mentioning. From the very beginning, Sencha Touch has implemented Sass to allow easy style changes to the UI. This means if you want to change your entire color scheme, you can do so with relative ease if you know Sass.

LEARN MORE ABOUT SASS

Sass has taken the world of style sheet management by storm and has arguably revolutionized how people style their web pages and apps. To learn more about this utility, check out Sass in Action at http://www.manning.com/netherland/.

Sencha Touch includes an MVC framework that will allow developers familiar with that pattern to develop applications within a familiar workspace. It also contains a custom URL routing mechanism and history state support.

The widgets that users will interact with in your application comprise the UI portion of Sencha Touch. When thinking about designing and constructing your applications, there is a lot to choose from, which is why I think it's a good idea to look at each of them.

1.3 The Sencha Touch UI

The Sencha Touch UI is a rich mixture of widgets that can be displayed on screen for you and your users to interact with. Given the size of the UI palette, I've organized table 1.2 to help us grasp the groups of UI components.

After reviewing the groups, we'll dive deeper into each group and discuss the UI components in greater detail.

Table 1.2 Looking at the various groups of UI widgets available in the Sencha Touch framework.

Group	Purpose
Containers	Widgets that are designed for nothing more than managing other child items. An example of these types of widgets is the Tab Panel. Containers typically implement layouts.
Sheets	Sheets are generally any popup or side-anchored container and appear in a modal fashion, requiring users to interact with the sheet before moving forward. An example of a sheet is the Date Picker widget.
Views	Views are widgets that implement data Stores to display data. The List and Nested List are both views that implement Stores.
Misc	This collection of widgets range from Buttons to Maps to Media.

Now that we have a good overview of the widget groups, we can begin our visual exploration.

1.3.1 Containers

Containers in Sencha Touch do the heavy lifting when it comes to managing widgets inside of widgets. Container is what I like to call the workhorse of Sencha Touch applications as it offers the extreme configurability and flexibility. Containers can dock child widgets to their sides or render child widgets inside of its body.

To see what I mean, take a look at the figure below.

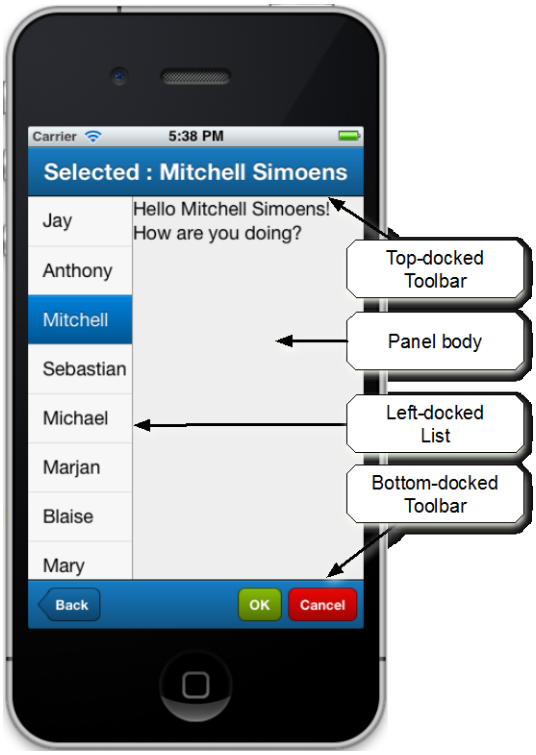

Figure 1.4 A demonstration of some of the Panel content areas.

In Figure 1.4 we see a `Container` with three docked items. We have a `Toolbar` docked at the top, `List` view docked on the left and another Toolbar docked at the bottom. Notice how the top-docked `Toolbar` simply contains a title, while the bottom toolbar contains buttons. This shows some of the power and flexibility of the `Toolbars`.

If you have a need to display screens controlled by a toolbar, the `Tab Panel` is what you'll need to get the job done.

1.3.2 Controlling your UI with Tab Panel

`Tab Panel` is a container well and automatically sets a top-docked or bottom-docked toolbar for you with automatically generated buttons for every child item. Tapping any of the buttons allows you to "flip" through items known as "cards". For example, see figure 1.5 below.

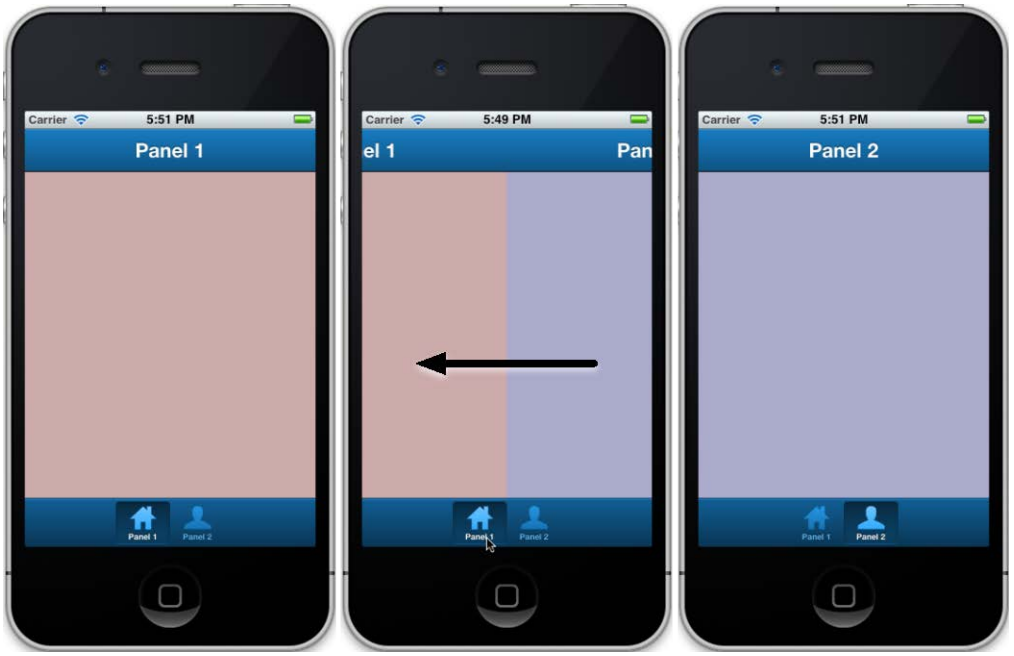

Figure 1.5 The Tab Panel allows you to configure UIs that can be changed by a tap of a button and includes optional transition animations (from left to right).

In the figure above, I've configured a `Tab Panel` that implements a "slide" transition with two child panels. By selecting Panel 2 in the `Toolbar`, Sencha Touch automatically applies the CSS3 transition properties to both child panel elements, allowing for a smooth transition from one `Panel` to another.

The `Tab Panel` does an excellent job of managing the display of items in your screen, but sooner or later, you'll need the ability to accept data input from your users. Here, the Form Panel is what you'll use.

1.3.3 Accepting input with Form Panel

The `Form Panel` is a container that is typically used to display any of the input fields that Sencha Touch provides and is automatically scrollable. Your fields can be grouped via the `FieldSet` widget.

Below is an example of a `Form Panel` requiring user input.

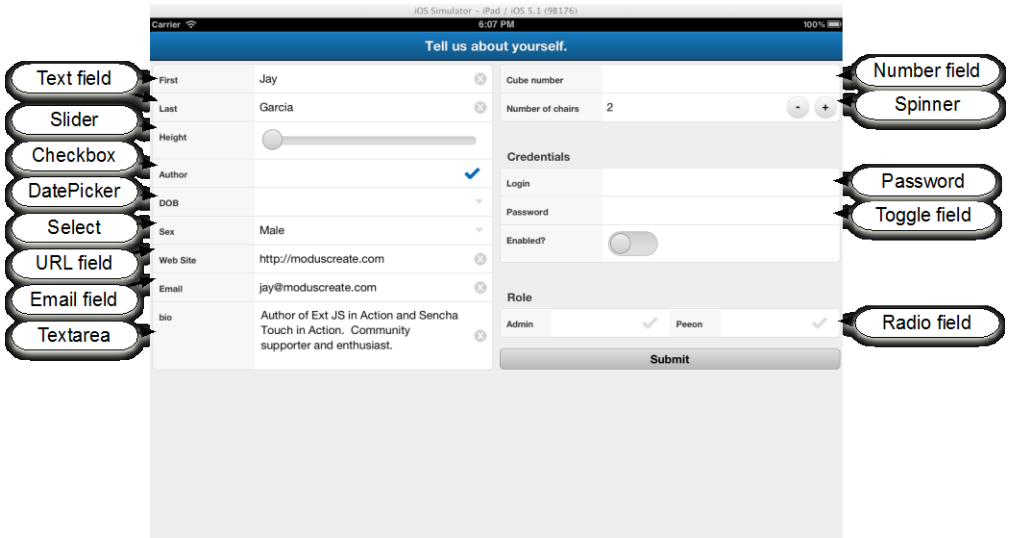

Figure 1.6 Form Panels are used to display input fields and contain necessary controls to manage the submission of data to your server.

In Figure 1.6 (above), we have most of the input fields that Sencha Touch offers, with the exception of the `Hidden` field. With Sencha Touch, the `Text`, `Checkbox`, `URL`, `Email`, `Textarea`, `Number`, `Password` and `Radio` fields all implement native HTML5 input elements, with the addition of styling. Each of these, minus the `Radio` and `Checkbox` fields, will force the native slide in keyboard to appear when focused, allowing users to enter data into the fields.

The Checkbox and Radio fields work similarly to their native-web counterparts, except they are stylized via Sencha Touch's own check icon to mimic native application behavior. In this example, the Role checkboxes, grouped in a `Fieldset`, are `Radio` fields, allowing only one selection in the set.

Next, the `Spinner` field is a custom styled input field, allowing users to enter numeric values, much like the `Number` field, with the addition of easy-to-use decrement and increment buttons on each side of the field.

The `Slider` field implements native Sencha Touch `Draggable` and `Droppable` classes, allowing users to input a numeric value, via a swipe and tap gestures. The `Toggle` field extends `Slider`, allowing users to toggle a field from two values via swipe and tap gestures, much like an on/off toggle switch that you see in various physical devices.

Lastly, the `Date picker` field and `Select` fields give your users the ability to choose data from a set. The `Date picker` field implements what's known as a Sheet, which is an overlay panel that slides in from the bottom, allowing the user to select values via vertical swipe or "flick" gestures.

Below is an example of a `Date picker` displayed in an iPhone.

Figure 1.7 The Date picker slider in action.

No matter what the device or its orientation, the `Date picker` field will always display a sheet forcing selection though this modal overlay. At this point, the `Date picker` widget should seem familiar to you, as it mimics the native iOS Date Picker input widget.

The `Select` field, however, will display different input widgets, based on the device. Below is an example of our implementation of the Select widget in a phone and tablet devices.

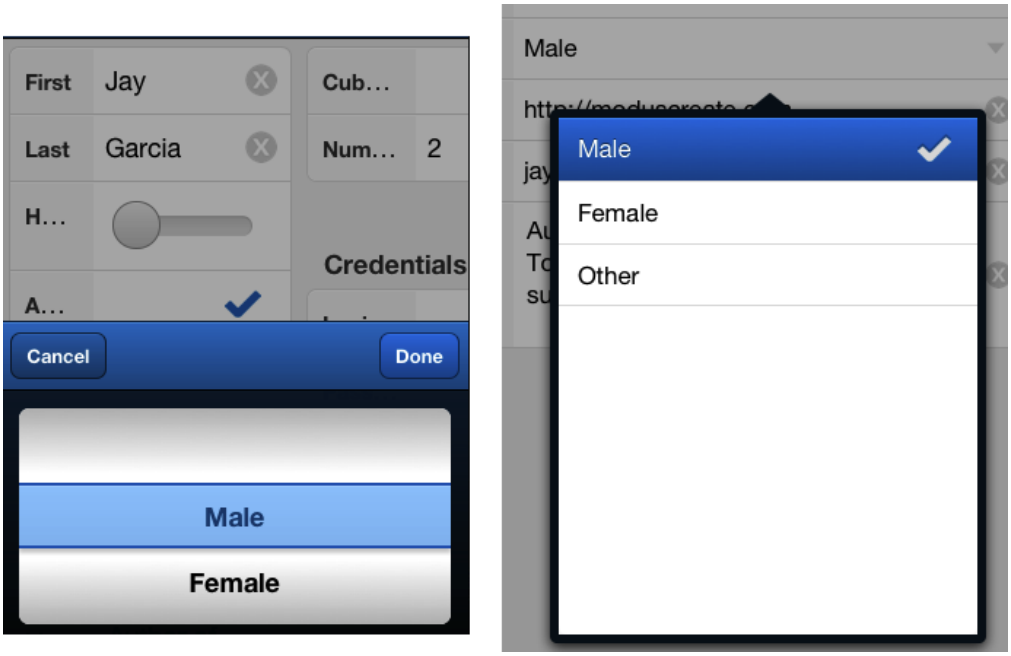

Figure 1.8 The Select field will display a different input widget based on the type of device that is running your UI.

In the left portion of the illustration above, the `Select` widget is displaying a `Picker` sheet because Sencha Touch detected that it's running inside of a phone versus a tablet (right). The difference has to do w/ the fact that iOS tablets natively display the dialogue-type of controls for selection.

As we've just seen, Sencha Touch offers quite a lot of wrapped native HTML5 input fields, as well as a few custom widgets. Since we've been talking about the Date picker and Select field implementing Sheets, we should take a look at the various Pickers and Sheets that Sencha Touch offers, outside of the `FormPanel`.

1.3.4 Sheets and Pickers

We've already seen the `Date picker` and Picker classes implemented via their associated form input widgets. Sencha Touch provides you with a widget called Sheet, which is a floating modal panel that animates into view, grabbing the user's attention and focus.

Below is an example of a `Sheet` in action.

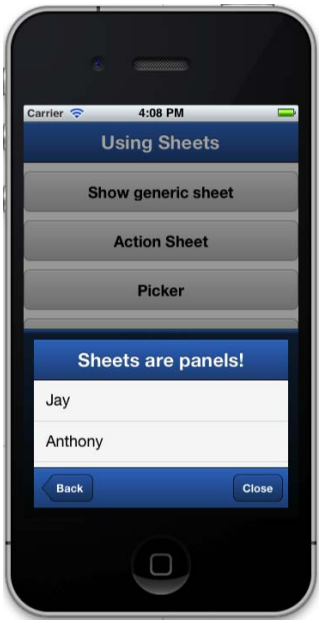

Figure 1.9 A generic Sheet in action displayed in portrait mode.

In the above illustration, we are displaying a Sheet with top and bottom-docked Toolbars, managing a scrollable `List` view. We can configure such a UI because `Sheet` is actually a subclass of `Panel` and brings forth all of the UI goodness that it provides.

What's neat about the `Sheet` widget is that it's orientation aware. This means that flipping the device while the application code above is executing immediately causes the sheet to render in landscape mode, as illustrated below.

Figure 1.10 A generic Sheet displayed in the landscape mode.

The story about Sheet does not end here. It has three subclasses, which are `ActionSheet`, `MessageBox` and Picker. Now, we've already seen `Picker` and its subclass, `Date picker`, but we have not seen `ActionSheet` and `MessageBox`.

I know that I'm throwing a lot of new names at you, so to help with any confusion, I've included a simple inheritance model diagram.

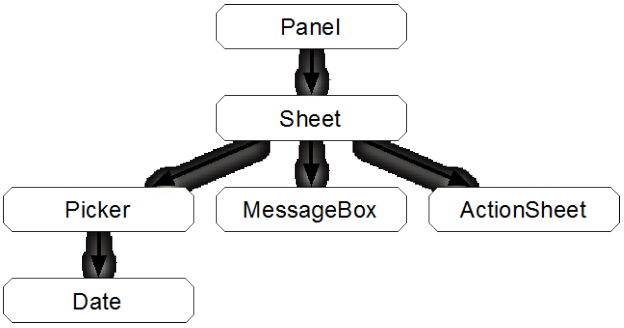

Figure 1.11 The Sheet inheritance model.

Out of the box, the `ActionSheet` widget allows you to easily render buttons in a sheet, rendered in a vertical stack. Because `ActionSheet` is a `Sheet`, which extends `Panel`, we can add pretty much anything we want to the stack of widgets.

Here is an example of an `ActionSheet` rendered with a custom HTML title.

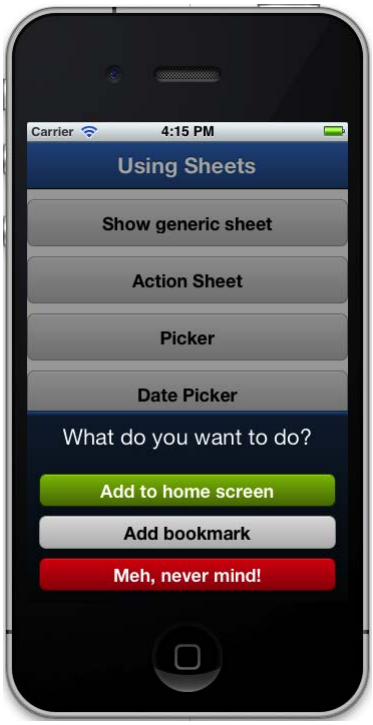

Figure 1.12 An ActionSheet displayed in an application.

Such an `ActionSheet` could be used to request action from the user, requiring that they choose an action via one of the buttons. In this case, we're asking the user to choose one of three options, and with the custom title, are providing a hint along with the actionable button set.

The `MessageBox` widget is a subclass of Sheet that provides Sencha Touch styled alert-like functionality to our applications. Here is an example of the three most common `MessageBox` display patterns.

Figure 1.13 The common MessageBox implementations alert (top left), confirm (bottom left) and prompt (right).

The illustration above displays the three most common uses of MessageBox, including alert (top left), confirm (bottom left) and prompt (top right). Each of these dialogues appear with smooth CSS3 transitions and mimic their native counterparts.

The key difference between the three is apparent. The alert `MessageBox` is designed to alert the user of some condition and only displays one button. The confirm dialog actually allows the user to make a decision by tapping on a button, allowing for a branch of logic to execute. Lastly, the prompt `MessageBox` asks the user for direct input.

We've seen all that Sheet and its subclasses have to offer. Next, we'll look at the various data-bound views that Sencha Touch offers.

1.3.5 Data-bound views

If you're an Ext JS developer, you might be surprised to learn that Sencha Touch only provides three data-bound views and that this list excludes a `GridPanel`. What we have at our disposal is a `Data View`, `List` and `NestedList`. We'll begin with the most basic, and work our way to the most complex, starting with the `Data View`.

The `Data View` is a widget that binds to a data Store to render data on screen. It gives you 100% control on how you will render your data. Below is an example of a simple `Data View` displaying a set of names, beginning with the last name.

Figure 1.14 A stylized Data View with an "itemtap" event handler, displaying an action sheet based on selection.

Above, we have a stylized `Data View` rendering data from a Store, which contains a list of names. Now, I just rendered names to keep it simple, but `Data Views` can be used to render pretty much anything imaginable, and allow for user interaction.

With the `Data View`, we're completely responsible for a lot of work, including defining the `XTemplate` that will be used to stamp out the HTML fragments, and styling how the items are rendered on screen. If you're like me and want the look and feel of a native list, the `List` class is available at your disposal.

Here's a List widget, rendering the exact same data we have above.

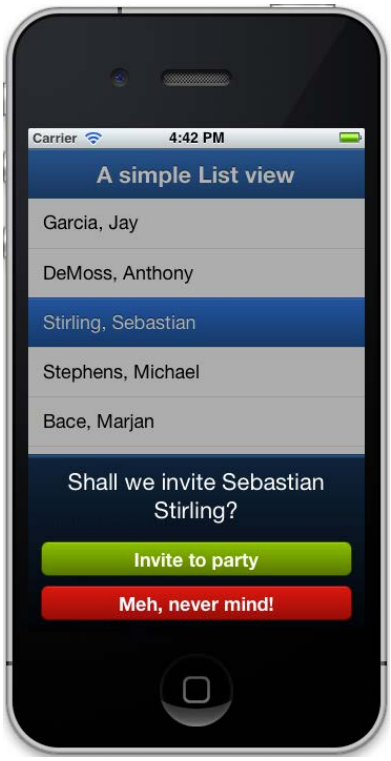

Figure 1.15 A simple List View with an action sheet.

Illustrated above is a `List` view widget rendering the same data that we did in the `Data View` example we looked at previously. The difference between the two is pretty obvious. What is not so obvious is that the level of effort to create this list view is orders of magnitude less than creating the previously viewed `Data View`. You'll see what I mean by this in Chapter 6, when we tackle List views head on.

There are three more key features that the List view provides in addition to the native application look and feel, and those are illustrated below for further discussion.

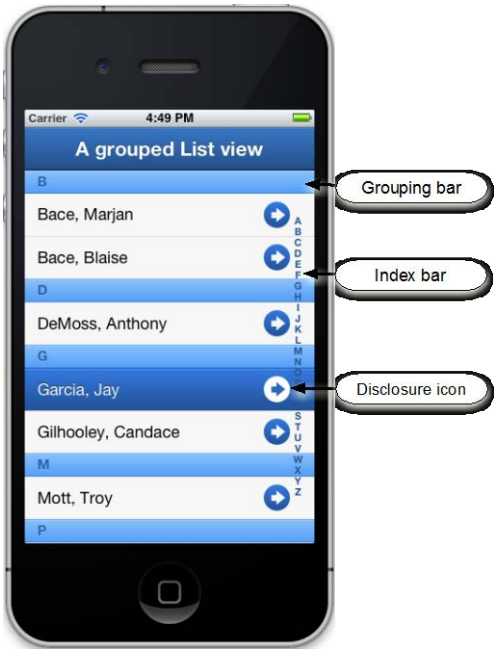

Figure 1.16 A grouped list view, sporting an Index bar.

With a few minor tweaks, we are able to transform a simple list view into a grouped `List` view. The grouped List in the illustration above has what is known as a grouping bar, which is a separator between items in the list. The Sencha Team has been able to get this list to work nearly identically to native grouped lists, and includes optional disclosure icons, as well as an index bar, for fast searching with a single finger swipe.

The `Data View` and `List` widgets are designed to display data in a linear set. However, there are times where you want to display nested data. For that, you'll need to use the `NestedList` widget.

Here is a NestedList widget in action.

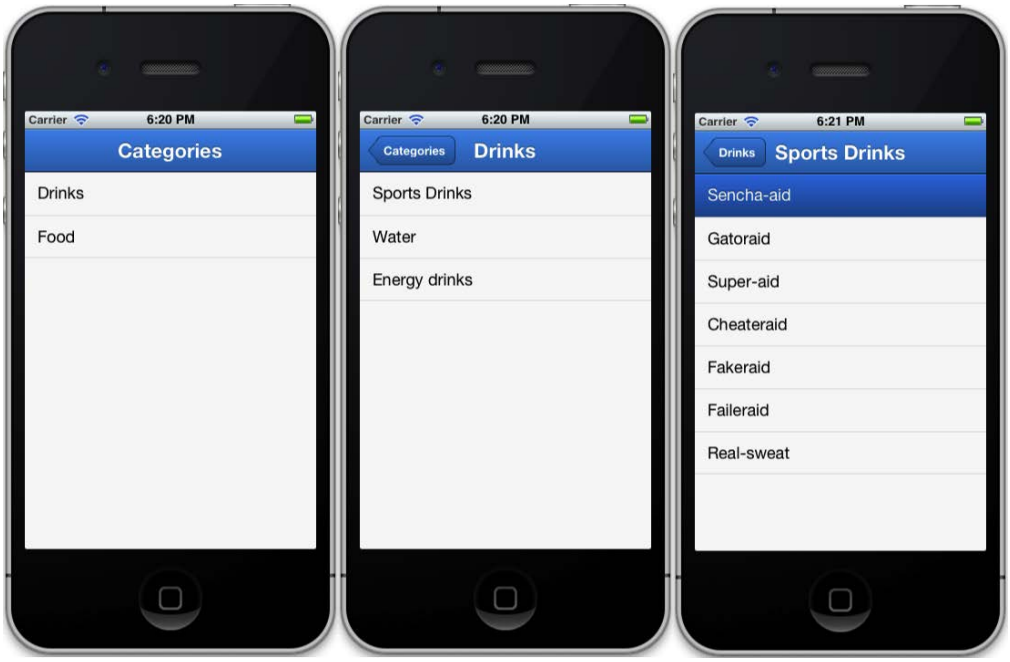

Figure 1.17 A NestedList in action used in a navigational manner.

In the illustration above, I have setup a NestedList for the selection of a food item. There are two main categories, Drinks, and Food. I chose "Drinks" (left), which brought me to three sub-categories. I then chose "Sports Drinks" (center), which led me to the last section of items, which is a list of sports drinks (right). All of this is done with the slide animation.

We've just explored the world of the `Data View`, `List` and `NestedList` widgets. Next, we'll explore the world of the Map and Media widgets.

1.3.6 Maps and Media

With the rapid expansion world of mobile applications, having the ability to integrate maps in your applications can be a huge boost in productivity for your users. To meet this growing demand, Sencha Touch integrates Google Maps to supercharge your location-aware applications.

Below is a screenshot of the Sencha Touch `Map` widget in action.

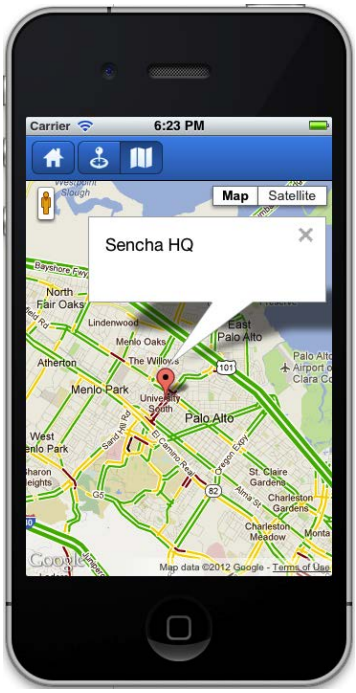

Figure 1.18 The Sencha Touch Map component in landscape mode.

The Sencha Touch `Map` widget literally *wraps* Google Maps, allowing your application code to manage the Google Maps instance as if it was native to Sencha touch. This means that the `Map` widget can take part in layouts, and has normalized events, as well as an interface method to easily update the map's coordinates.

Another growing demand for mobile applications is the ability to play audio and video content. HTML5 natively has video and audio tags that bring this functionality to Mobile Safari, but Sencha Touch makes it easier to use.

Below is an example of the Media widget displaying a video on a tablet.

Figure 1.19 A Panel wrapping a Media widget to play video on a tablet.

Just like the Map widget, the Media widget brings familiar interface methods, and easy configurability to play audio and video in your applications.

We've just completed our UI walkthrough. Before we wrap up this chapter, I want to talk to about thinking like a Mobile developer. This conversation will be especially helpful to you if this is your first dive into the world of Mobile application development. I know you're itching to get down to coding, so I won't hold you very long.

1.4 Thinking like a mobile developer

If you're like me, you're making the transition from Ext JS to Sencha Touch. However, making the transition to mobile from desktop application development from a thought-perspective poses challenges that must be overcome if you are to build successful apps.

Even though Sencha Touch is a relatively new framework, there are some things that immediately come to mind when making the transition from desktop to mobile. Here are some points you need to think about before moving forward with your application development.

1.4.1 Think light-weight

When spec'ing out or developing your mobile app, you must think "lightweight" or your app is destined to run into performance issues. If you've made the transformation from a native-desktop application developer to a desktop-web application developer, it is likely that you've hit this issue during that transition, where native desktop applications can handle much more of a burden than desktop-web applications.

Due to the reduced computing power of mobile devices, the mobile browser is limited in many ways, when compared to its desktop counterpart**.** This is why thinking "lightweight" is paramount for a successful application.

My suggestion is to try to reduce the amount of data as well as the complexity of the screen size as possible. Reducing the user interaction models is also a plus, as complex user interaction models for mobile web applications.

1.4.2 Remember - it's a browser!

Many developers are tasked with converting native applications to Sencha Touch-powered web applications. Often times during the conversion process, performance issues are hit and Sencha Touch is blamed.

It is during these times that I tell developers who are caught in this cycle to remember that the application they are developing *is* running inside of a browser, thus has limited power relative to native-compiled mobile applications. Just like native-desktop applications can handle more difficult tasks when compared to desktop-web applications, native-mobile apps have more muscle when compared to mobile-web applications.

I believe that entering the conversion process with this in mind helps you truly set realistic expectations with your customers.

1.4.3 Throw away what you don't need

With the reduced power of mobile devices comes an increase of responsibility to keep things as clean as possible, and reduce DOM clutter/bloat. For desktop-web applications, this is not so critical, but for mobile-web it is extremely critical.

This means that when placing items in the DOM, such as `ActionSheets`, you must take the care to destroy them. DOM bloat is the enemy of performance.

> **MOBILE SAFARI WILL CRASH**
>
> Mobile Safari will crash if your application causes it to run out of memory. This will simply cause the application to disappear from the user, without warning.

Sencha Touch widgets come with a complete three-phase lifecycle, allowing you to easily destroy components, thus removing items from the DOM and freeing up crucial resources.

Along with the destruction of items that are no longer needed is the thought of only instantiating what is truly needed. I often find hugely nested components being instantiated when only a single component is needed for a particular action. To keep things safe, try to think conservatively.

1.4.4 "finger" !== "mouse"

Part of transitioning to mobile development is the understanding the user interaction models and how they relate to browser events. The following table describes some of the most common user gestures, alongside their desktop counterparts (when applicable).

Table 1.3 Comparing touch gesture events with desktop mouse gesture events.

Mobile	Desktop	Description
tapstart	mousedown	The initial point at which a touch is detected in the UI.
tapend	mouseup	Signals the end of a touchstart event.
tap	click	A tapstart and tapend event for a single target.
doubletap	doubleclick	Two successive tapstart and tapends for a single target.
swipe		A tapstart and tapend events with a delta in X (horizontal) or Y (vertical) coordinates.
pinch		A complex multi-finger “pinch” gesture. It is comprised of multiple touchstart and touchend events with deltas in the X and Y coordinate space.

Always try to test all of your complex interaction models with the physical platforms that you are targeting for your applications. It’s only then that you can truly see how it will react during events like pinch, swipe and drag.

1.4.5 Reduce the data

When developing your applications, you have to remember to reduce the amount of data you’re sending to the browser. If you find yourself pushing megabytes of data to the server for a single Ajax request, you should reconsider your approach.

Along these lines is the reduction in data complexity. Remember that these devices are relatively low powered, and any time spent manipulating complex data could be spent allowing the user to interact with the application. Tasking your mobile application to deal with deeply nested and complicated data structures is highly discouraged.

> **Server-side developers will have to work harder**
>
> Lots of times deeply nested data structures are passed to clients because of the amount of work it is for the server-side developers to reduce complexity. I’d much rather have our server-side developers work harder than impact the performance of the client, thus shedding negative light on our mobile applications.

Through your application development iterations, I suggest often placing yourself in the shoes of the end-user. Remember, mobile applications should be quick and responsive.

1.5 Summary

We covered quite a bit in this first chapter, beginning with a high-level discussion about what Sencha Touch is and the problems that it aims at solving for the mobile-web application space.

We then took a deep dive into the world of the SenchaTouch UI widget set and got to learn about hat is offered out of the box. Along the way, we identified some of the differences between widgets, such as the Data View and List view.

Lastly, we discussed some of the ways you should think about mobile applications and some of the limitations that mobile devices pose.

In the next chapter, we're going to begin our deep dive into Sencha Touch, beginning with where to get the framework, and inspect its contents. After we've become familiar with setting up a basic Sencha Touch app page, we'll develop a quick application with the framework.

2

Using Sencha Touch for the first time

This chapter covers

- Downloading Sencha Touch
- Looking at the package contents
- Creating a simple “Hello World” example
- Developing a simple application

In the last chapter, we looked at what Sencha Touch provides us developers. We effectively set the stage for us to begin using the framework in this chapter. That is precisely what we will be doing.

In this chapter, you’ll begin to understand how pieces of the Sencha Touch work and can fit together as we work to develop an application using critical pieces of the framework such as the class Loader. Since this is an introductory chapter, we will not be going into too much detail of how things operate. That level of detail will begin starting with Chapter 3.

2.1.1 License considerations

To get a copy of the Sencha Touch framework, you will need to visit http://sencha.com/products/touch/download/. You’ll be greeted with a page like the one in Figure 2.1.

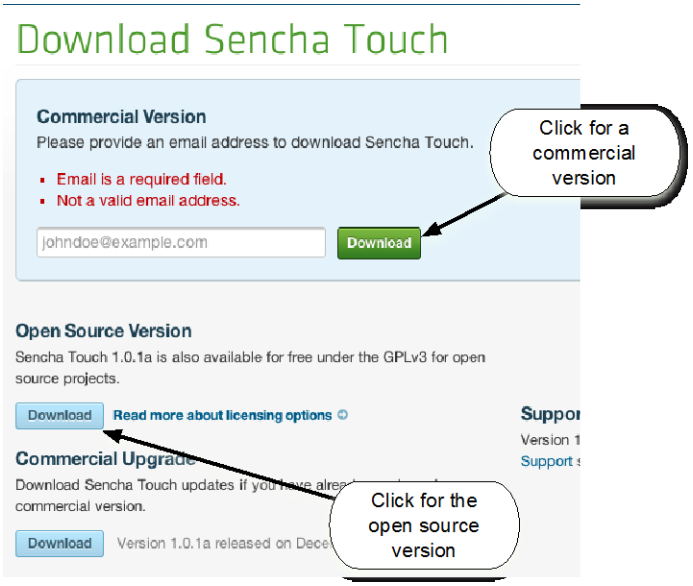

Figure 2.1 You must make a decision on which version of Sencha Touch you're going to download.

From here you're required to make a decision on which license you're going to use. If you're not sure which version to choose, I've compiled a few scenarios to help you with your decision.

Table 2.1 Choosing a license model for Sencha Touch.

License Type	Reason
Open Source	Your project will be available for anyone to download and modify according to the GNU GPL license v3. There are FLOSS (Free Libre and Open Source) exceptions available.
Commercial	You are creating an application that you plan on selling or distributing and not willing to permit direct access to your source. If you plan on packaging your product for offer on an app store, this is the license you're going to want to go with.

If you are still unsure of which license to choose, you can consult the Sencha licensing page at http://sencha.com/products/touch/license or email the Sencha license support team at licensing@sencha.com.

Now that we've gotten the hairy licensing issue out of the way, we can begin to look at the package contents of the framework.

2.1.2 Unpacking the framework

The Sencha Touch framework comes to you via a Zip file, which requires unpacking in order to actually use the library. Here's what you can expect coming out of the zip file you downloaded.

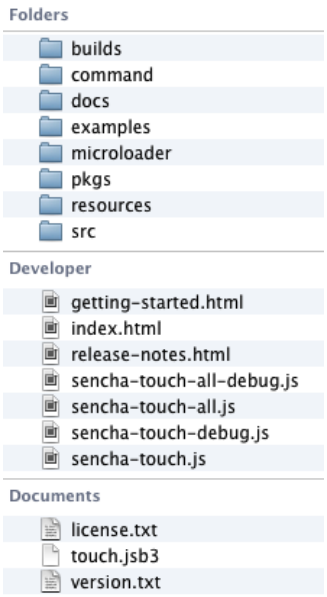

Figure 2.2 Looking at the package contents.

The zip package inflates to a directory that is just over 56 Megabytes. If you're wondering if all of the package contents is required to be deployed with your application, the answer is no.

The good news is that most of the space used in the package goes to docs, examples and other goodies that you typically don't and shouldn't deploy with your applications. Here's a description of each of the items in the archive that you just extracted.

Table 2.2 The various files in the Sencha Touch package.

Item(s)	Purpose
builds/	Contains sencha-touch-compat.js file, which contains the code you'd use if you wanted to run your app with backwards-compatible 1.0 code. I highly advise not depending on this version of the framework for your application code. I suggest migrating to 2.0 as quickly as possible.

command	This directory contains necessary files for the Sencha Command toolset, which is like a swiss army knife of the Sencha Touch 2 development world. We won't be diving into Command until we talk about advanced topics in Chapter 9.
docs/	The docs directory is where you'll be able to view the framework documentation without the need to go online. This directory itself is over 46MB in size.
examples/	This directory contains all of the one-off examples as well as a few example applications. This directory is also responsible for the hefty weight of the unzipped package, as it's over 29MB in size.
pkgs/	When Sencha Touch is "minified" (reduced in size) by JSBuilder, it creates what are known as "packages" or segmented groups of features. Typically, you won't use this directory at all.
resources/	This folder contains all of the CSS and image files required to style the framework. It also has a subdirectory named "sass", which includes all of the SASS code needed to get you off to developing your own custom themes.
src/	Here is where all of the complete source files are located. You would typically view the src directory contents if you want to take a deep dive into the framework and learn how the widgets work.
getting-started.html & index.html	getting-started.html is a rather lengthy write-up on how to get started with Sencha Touch, while the index.html file is to serve as a landing page for you if you ever browse the contents of the framework directory. From here, you can launch the API documentation and example pages.
release-notes.html	This is a document that describes, in great detail, all of the changes to the framework since its initial release.
sencha-touch*.js	There are three "sencha-touch" js files included. Each with varying sizes. I typically use 'sencha-touch-debug-w-comments' when debugging locally inside of a browser and use 'sencha-touch-debug.js' when debugging on a device. The comments file contains extra weight that makes it 1.54MB in size, while the debug file without comments is only over 750K. When going to production, you should be building a custom version of Sencha Touch via Sencha Command. This will give you an optimized build based on your application needs.

As you can tell, there is a lot in the Sencha Touch framework out of the box. When it comes to my dev environments, I typically remove a lot of the extra stuff like pkgs, examples, docs, and any unused versions of Sencha Touch. For docs and examples reference, I simply refer to a virtual host on my dev machine, which contains each version of the framework for reference.

Being that we have a good understanding of the package contents, we can begin with setting up Sencha Touch on a dev web server.

2.2 Sencha Touch says "Hello World"

To get this party started, we're going to need to create an html file that we will have to populate with HTML and required JavaScript and CSS references. Here is what my test page looks like.

Listing 2.1 The HTML body for our helloworld.html file.

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <link rel="stylesheet"                                                    #1
          href="js/sencha-touch-2.0.0/resources/css/sencha-touch.css"
     type="text/css">

    <script type="text/javascript"                                            #2
            src="js/sencha-touch-2.0.0/sencha-touch-debug.js"></script>

    <script type="text/javascript"></script>                                  #3
</head>
    <body>
    </body>
</html>
```

{1} Include Sencha Touch CSS
{2} Reference the Sencha Touch debug JavaScript
{3} An empty script tag for our use.

In listing 2.1, we have all of the necessary structured HTML for our simple hello world page. It includes the required tags for the Sencha Touch CSS**{1}** and JavaScript**{2}** files as well as an empty script tag **{3}** that we can use as a simple playground.

If you load this page now, you're only going to see a blank page. That's because we have not implemented anything from Sencha Touch yet.

Here is some code that will launch the Sencha Touch MessageBox, providing indication that everything is working well.

Listing 2.2 Using Sencha Touch for the first Time.

```
<script type="text/javascript">
    Ext.setup({                                                               #1
        onReady : function() {                                                #2
            Ext.Msg.alert(
                'Hello!',
                'Hello there from Sencha Touch!'
            );
        }
    });
</script>
```

#1 Calling Ext.setup for the first time

#2 Our Ext.onReady function

In Listing 2.2, we use Sencha Touch for the first time, rendering the `MessageBox` in our screen. Here's how all of this works.

To bootstrap our code, we execute `Ext.setup` **{1}** inside of our playground JavaScript code block. `Ext.setup` is a utility function that is used to configure Sencha Touch for use on your mobile devices and should only be called once, as it is designed to assist with bootstrapping of your code if you do not intend to utilize the Sencha MVC framework.

We pass one parameter to `Ext.setup`, which is known as a "config[uration] object". In this config object, there is one property, known as onReady **{2}**, which is executed by Sencha Touch when it detects that Webkit has the document fully loaded and is ready for direct manipulation.

> **Know thy config object.**
>
> Since you will see the term "config object" used a lot in the Sencha Touch API documentation and this manuscript, it's important that you know exactly what we're talking about. Like Ext JS, in Sencha Touch most constructors and some methods use config objects, which are used by said classes to modify how the class or widget behaves.

Inside of the `onReady` function is our use of the `MessageBox` class, accessible via the `Ext.Msg` singleton. To get the `MessageBox` widget to display, we call the `alert` method and pass three parameters, title, message body and callback. The callback is simply a reference to a reusable empty function that resides inside of the Ext namespace.

> **EXT NAMESPACE?**
>
> Sencha Touch uses the Ext namespace because it shares a common codebase as its big brother, Ext JS. This common codebase is known as the Sencha platform. If the Sencha Touch framework code were to be moved to a "Sencha" namespace, some unnecessary weight would be added to the framework.

Our helloworld.html file is all ready to be viewed in a browser or device. Figure 2.3 illustrates our first Sencha Touch code execution.

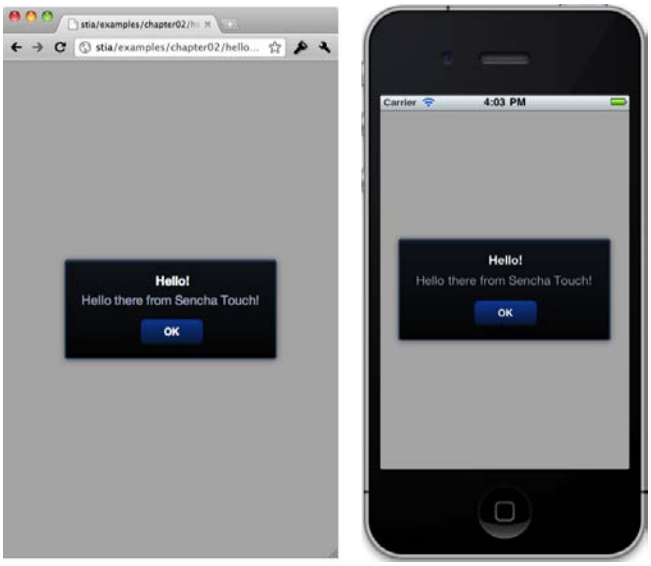

Figure 2.3 Our helloworld.html in a browser and phone.

Success! We have our first Sencha Touch code live and working. As illustrated in Figure 2.3, our code is executable in a Desktop browser as well as a mobile device. In this particular instance, our code is running inside of Chrome.

YOUR DESKTOP BROWSER CHOICES

Because Sencha Touch is designed to work with Webkit browsers only, the choice of desktop-based development and debugging tools are limited to two browsers, Safari and Google Chrome. Both of these browsers are based on the Webkit engine, but are somewhat different. At the time of this writing, Safari's 3D CSS is fully supported by Sencha Touch, while Chrome's 3D CSS currently in experimental phase and not fully functional.

Great! We now have our hands a little dirty with Sencha Touch, but we're not done yet. This simple demo was just so we can ensure that our first Sencha Touch project was configured correctly and functioning properly.

Before we move on to developing our simple application, I think we should revisit `Ext.setup`, as there are some key configuration properties that you should be made aware of before moving forward.

2.2.3 Learning more about Ext.setup

Recall that `Ext.setup` is a utility method to configure Sencha Touch and serves as a launchpad for your application. When we created our Hello World example, we only passed in an `onReady` configuration item. There is a lot more that Sencha Touch offers you with `Ext.setup`. Table 2.3 lists out all of the popular configuration properties and their usage.

Table 2.3 The various Ext.setup configuration options

Config Poperty	Purpose
fullscreen	Set this option to Boolean true to allow your application to execute in full screen mode. Doing so instructs Sencha Touch to initialize a special and private class called the Viewport to make full screen mode possible in a cross-platform manner. The Viewport class contains the basic plumbing to manage orientation handling and browser resizing.
tabletStartupScreen	If you want a startup screen to display when your application bootstraps, you will want to set this configuration option to the URL reference of that image. You will need to consult your hardware manufacturer for details on the screen size for your target devices.
phoneStartupScreen	Set this option the URL reference of an image that you would like to use as a startup screen for your phone. Since device resolutions vary, you should consult the manufacturer documentation for details on the screen size.
icon	This option should be set to the URL reference of your 72x72 pixel icon. It should only be set you plan on using the same icon for phones and tablets.
tabletIcon	If you plan on having a tablet-specific icon, you want to set this property to the URL reference for that icon. Your icon should be sized to 72x72 pixels. This property will override the `icon` property.
phoneIcon	This configuration option is used to set the URL reference for a phone-specific icon for your application. Icons should be sized to 57x57 pixels. This property too will override the `icon` property.
glossOnIcon	Set this configuration option to Boolean true to instruct iOS to add a gloss effect to the home screen icon for your app.
statusBarStyle	This configuration option is used to configure the iOS top-most status bar style. There are three available options to you, which are “default’, ‘black’ or ‘black-translucent’.
onReady	This is the function that is to be executed as soon as Sencha Touch detects that the Webkit browser is ready to be manipulated.
scope	This configuration option sets the execution scope of the onReady function.

As we just learned, there are quite a lot of important `Ext.setup` properties that we should be aware of. The reason there are so many configuration properties is to allow as much flexibility as possible for configuring your application for various target platforms.

We just got our feet wet with Sencha Touch and took a deep dive into `Ext.setup` configuration options. Doing all of this work sets the stage for us to develop a basic application with this framework.

2.3 Setting the stage for our first application

Since this is most likely your first attempt at creating a Sencha Touch application, we're going to take things easy. For instance, we will not be worrying about any target devices or orientation or even MVC.

We will also not implement any server-side code. Instead, we'll use an existing API online and fetch data via JsonP. Before we can write a single line of code, we need to discuss what the application is actually going to do.

2.3.1 Our simple application at a glance

Today, we're going to build a relatively simple contact manager application. This application will allow us to view details of records and have a left to right workflow. Here's what it will look like in a tablet.

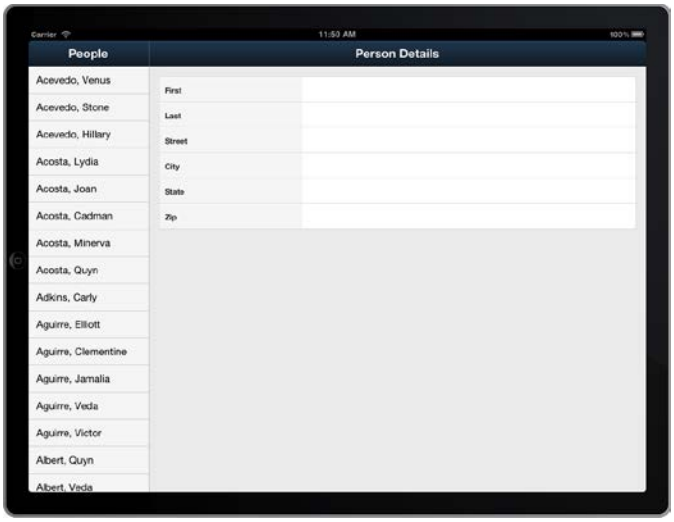

Figure 2.4 A preview of our contact manager app.

Looking at Figure 2.4, we can see that we'll be implementing a List and FormPanel widgets, along with some Toolbars. Our app will work in a typical left to right workflow, where we'll be able to select a record on the left, and see its detail on the right in the form Panel instance.

Our next step is to prepare a project to allow us to construct this application.

2.3.2 Preparing our project

To construct the foundation of our code, we'll need to first create the some files and a directory to organize our class structure. We'll be creating these separate files because I want you to get use to the idea of having one class-per-file, as it is considered a best practice by the Sencha community and follows OOP.

We'll be using the Class loader system, so you'll need to create some files and folders in your development environment like Figure 2.5. For now, let's just make these empty files. We'll be filling them out along the way.

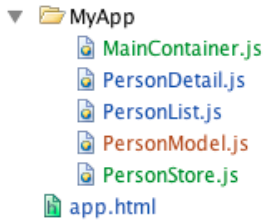

Figure 2.5 The directories that implement the MVC code structure.

In our example directory, we have one directory, MyApp with several files. Inside of the MyApp directory are the five JavaScript class files we'll be soon creating.

Here's a quick list of what function each file will perform.

Table 2.4 The list of files in the views directory along with their role in our application.

File	Role
MainContainer.js	This file will extend Ext.Container and play as a home for the PersonDetail and PersonList class using the VBox layout.
PersonDetail.js	Here, we'll extend Ext.form.Panel and render a few text Fields, displaying the details of the PersonModel instances selected from the PersonList class.
PersonList.js	We'll be extending Ext.dataview.List here to display the data in the data.json file and implementing the PersonStore class.
PersonModel.js	This class will define our data model for the person, and provide the PersonList and PersonDetail widgets with the data for each record in the data.json file.

PersonStore.js	Here, we'll extend Ext.data.Store and implement our PersonModel class. We'll be configuring this class to pull data from an online resource.

As siblings to the MyApp directory, we have app.html, which will launch our application.

That's pretty much all there is to it. We now have a clear picture of what we'll be constructor. We can start to get our hands dirty.

2.4 Developing Our Simple App

We'll begin developing our simple application by defining the Model (PersonModel) that will drive our data store extension, PersonStore. All of our classes will be inside of the MyApp namespace.

We'll be using the class Loader system, which means that we'll have to setup a decency model, where one class will cause the automatic loading of another.

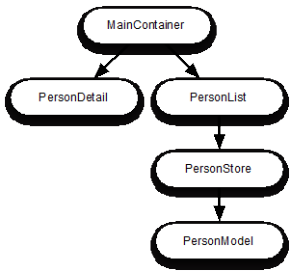

Figure 2.6 Our class dependency model.

EXPOSURE TO ADVANCED CONCEPTS

We're going to be using Ext.define, a method that allows us declare classes. It's OK if you don't fully understand what's going on here. But, if you want to learn about Ext.define take a peek at chapter 9. But come right back when you're done. We have a lot of work to do.

Let's begin by creating the PersonModel class. The following code will reside inside of MyApp/PersonModel.js.

Listing 2.3 Our PersonModel extension.

```
Ext.define('MyApp.PersonModel', {                    // 1
    extend : 'Ext.data.Model',                       // 2
    config : {
        fields : [                                   // 3
            'city',
            'firstname',
```

```
            'lastname',
            'middle',
            'state',
            'street',
            'zip'
        ]
    }
});
```

{1} Define our model
{2} Extend Ext.data.Model
{3} Configure the data fields

To define our PersonModel class, we employ Ext.define, specifying the full namespace for our MyApp.PersonModel**{1}** extension. We instruct Ext JS to extend the Ext.data.Model class**{2}**, and configure the data fields**{3}** for the model to use. These will be useful in mapping the inbound data from our remote web service and will be used to supply data to the PersonDetail extension.

That wraps up the PersonModel class file. We can now move on to adding content to the PersonStore.js file.

2.4.1 Creating the data Store

Our next step is to define the data Store extension, which will implement our PersonModel class. Here is the code for the PersonStore, which will reside inside of MyApp/Personstore.js.

Listing 2.4 The PersonStore extension.

```
Ext.define('MyApp.PersonStore', {                                    // 1
    extend   : 'Ext.data.Store',
    alias    : 'store.personstore',
    requires : [ 'MyApp.PersonModel' ],                              // 2
    config   : {
        autoLoad : true,
        model    : 'MyApp.PersonModel',                              // 3
        proxy    : {
            type   : 'jsonp',
            url    : 'http://extjsinaction.com/dataQuery.php',       // 4
            limit  : 20,
            reader : {
                type         : 'json',
                rootProperty : 'records'
            }
        }
    }
});
```

{1} Define the PersonStore class
{2} Setup class dependency
{3} Implement PersonModel
{4} Use the external web data source

In Listing 2.4, we define our PersonStore class**{1}**, an extension to Ext.data.Store. After instructing Ext JS to extend the data.Store class, we configure an alias parameter, giving us the capability to

In order to have Sencha Touch automatically load the PersonModel class, we'll need to document it as a dependency to the PersonStore class, which we do with the requires property**{2}**.

Next, tell the PersonStore to implement the PersonModel, via the model config and we instruct it to use a JsonP proxy that will request data from our online web service. This data store will automatically load once instantiated.

We now have the data portion of our application code defined. We can now move to configure our list extension.

2.4.2 Constructing the PersonList class

To create our PersonList class, we'll have to extend Ext.dataview.List and will require and implement the PersonStore class we defined previously.

Here are the contents of the PersonList class.

Listing 2.5 The PersonList class

```
Ext.define('MyApp.PersonList', {                                    // 1
    extend   : 'Ext.List',
    xtype    : 'personlist',                                        // 2
    requires : ['MyApp.PersonStore'],                               // 3
    config   : {
        allowDeselect : false,
        itemTpl       : '{lastname}, {firstname}',
        store         : {                                           // 4
            type : 'personstore'
        },
        items         : [
            {
                xtype  : 'toolbar',                                 // 5
                title  : 'People',
                docked : 'top'
            }
        ]
    }
});
```

{1} Define PersonList
{2} Configure an XType alias
{3} Require PersonStore
{4} Lazy implementation of PersonStore
{5} A simple title bar

To define the PersonList class**{1}**, we extend Ext.dataview.Listand setup an XType alias**{2}** for lazy instantiation later on inside of the of the MainContianer class we'll be soon creating.

After the alias, we setup the requirement for the PersonStore**{3}** class and will instruct Sencha Touch to load that class immediately after PersonList is loaded and defined.

The last block of code, the class config object, contains properties that allow us to configure how each instance of this class will behave. For example, we set the itemTpl property as a string that contains tokens that will render the first and last names, as defined in our data Store's model. This is possible because we configure our PersonList class to use our PersonStore class via lazy instantiation.

WHY SO LAZY?

Lazy instantiation is a pattern that we use in the Sencha world in which we define a plain JavaScript object and it is used to configure an instance of some class. We talk about lazy instantiation in Chapter 3, where we discuss XTypes.

The items configuration property allows us to configure a Toolbar that will be docked to the top of our List extension, providing a nice area for a title.

Here is what it would look like if we rendered it on screen by itself.

Figure 2.7 The PersonList class rendered by itself.

We have our PersonList class complete. Next up is the PersonDetail screen.

2.4.3 Building the PersonDetail

Our extension the Sencha Touch form Panel class will be known as PersonDetail in our code base, and will be responsible for constructing all of the fields required to display the details of a record selected in the PersonList class.

Listing 2.6 contains the code for this new class. This will be the longest listing that we have in this chapter. This is simply because the input fields require a lot of code, relatively speaking.

Listing 2.6 The PersonDetal class

```
Ext.define('MyApp.PersonDetail', {                                    // 1
    extend : 'Ext.form.Panel',
    xtype  : 'persondetail',                                          // 2
    config : {
        items: [
            {
                xtype       : 'fieldset',                             // 3
                defaultType : 'textfield',
                defaults    : { labelWidth : 100 },
                items       : [
                    {
                        label : 'First',
                        name  : 'firstname'
                    },
                    {
                        label : 'Last',
                        name  : 'lastname'
                    },
                    {
                        label : 'Street',
                        name  : 'street'
                    },
                    {
                        label : 'City',
                        name  : 'city'
                    },
                    {
                        label : 'State',
                        name  : 'state'
                    },
                    {
                        label : 'Zip',
                        name  : 'zip'
                    }
                ]
            },
            {
                xtype  : 'toolbar',
                title  : 'Person Details',
                docked : 'top'
            }
        ]
    }
});
```

{1} Define PersonDetail
{2} Configure XType
{3} Configure input fields

In listing 2.6, we define the PersonDetail class**{1}** and its XType alias**{2}**. The longest section of this listing is defining the child items**{3}** for this extension. Here, we define all of the input fields to display the data and a Toolbar docked at the top of this class, which contains a simple title.

Figure 2.8 illustrates what this widget looks like rendered by alone with data inside of it.

Figure 2.8 The PersonDetail widget rendered on screen

Our PersonDetail class is now complete and ready to be stitched in. Next, we'll be creating the MainContainer class, which will allow us to tie everything together.

2.4.4 Setting up the AppPanel class

As I mentioned a bit earlier, the MainContainer class is going to be the master UI class for our mini-application. This widget is going to be responsible for rendering the PersonList and PersonDetail widgets on screen, and will tie in the simple left-to-right workflow using event listeners.

Listing 2.7 contains the code for this MainContainer class and has some of the most advanced Sencha Touch implementation code yet.

Listing 2.7 The AppPanel class layout

```
Ext.define('MyApp.MainContainer', {                          // 1
    extend   : 'Ext.Container',
    requires : [                                             // 2
        'MyApp.PersonList',
        'MyApp.PersonDetail'
    ],
    config : {
        layout : {
            type  : 'hbox',
            align : 'stretch'
        },
        items : [                                            // 3
```

```
            {
                xtype  : 'personlist',
                itemId : 'list',
                width  : 200,
                style  : 'border-right: 1px solid #999'
            },
            {
                xtype  : 'persondetail',
                itemId : 'detail',
                flex   : 1
            }
        ],
        listeners : {
            select : {
                fn       : 'onListSelect',                          // 4
                delegate : '> #list'
            }
        }
    },
    onListSelect : function(list, record) {                         // 5
        this.getComponent('detail').setRecord(record);
    }
});
```

{1} Define MainContainer
{2} Require PersonList and PersonDetail
{3} Implement PersonList and PersonDetail
{4} Listen to the PersonList select event
{5} Display data in PersonDetail

We first define our MainContainer**{1}** class in Listing 2.8. Since we'll be instantiating this class directly, there is no need to configure an XType alias for it. In order to have Sencha Touch automatically load our PersonList and PersonDetail classes for us, we must list them in the requires configuration block**{3}**.

Next, we implement our PersonList and PersonDetail widgets by means of lazy instantiation with their XTypes**{3}**. We configure each widget with a special itemId. This will allow us to do things like configure event listeners and gain references to these components later on.

In order to tie the selection from PersonList and load the selected record into the PersonDetail class, we configure a listeners object, which defines the event to listen to (select), what function to call (onListSelect) and which child to hook the event into (PersonList). This leverages the new power of the Sencha Touch 2 event system, where we can define event handlers via String representation of method names and use ComponentQuery selector syntax to guide event handler registration.

The onListSelect method will always be called within the scope of our instance of MainContainer. Therefore, we can use the use this.getComponent, a Container method, to gain a reference to the PersonDetail instance via itemId, and execute its setRecord method, passing in the model instance (record) as the only argument. All of this will cause the

PersonDetail class to bind to the instance of the record and display its data in the input fields.

We have all of our classes defined and ready to roll and render our app.

2.4.6 Rendering our application

To render our application we will need to add HTML and JavaScript to our app.html file. Here are the contents of that file.

Listing 2.8 The contents of app.html

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type"
         content="text/html; charset=utf-8">
    <title>Sencha Touch in Action CH2 app</title>
    <link rel="stylesheet"
         href="../../sencha-touch-2.0.0/resources/css/sencha-touch.css"
         type="text/css">
    <script type="text/javascript"
         src="../../sencha-touch-2.0.0/sencha-touch-all-debug.js">
    </script>

    <script type='text/javascript'>
        Ext.Loader.setConfig({
            enabled : true,                                          // 1
            paths   : {
                MyApp : 'MyApp'                                      // 2
            }
        });

        Ext.require([                                                // 3
            'MyApp.MainContainer'
        ]);

        Ext.setup({
            onReady : function () {
                Ext.create('MyApp.MainContainer', {                  // 4
                    fullscreen : true
                })
            }
        });
    </script>

</head>
<body>
</body>
</html>
```

{1} Enable Sencha Touch Loader
{2} Configure namespace and Path
{3} Automatically load MainContainer
{4} Render MainContainer in full screen mode

Listing 2.8 contains all that we need to render the entire application. Aside from the HTML that defines the page and loads Sencha Touch and the necessary CSS, we have a script tag that does a little bit of work to bootstrap things and finally render the application.

The first step to render our app is to enable the Sencha Touch class loader via the Ext.Loader.setConfig method call**{1}**. We also have to tell Loader what directory to look for the class files in our MyApp namespace**{2}** and do this via the paths property.

To render the application, we call Ext.require and pass our MainContainer class**{3}**. This will allow Sencha Touch to load this class using Script tags, which is considered best practice for performance and will delay the "ready state" for the browser. Because we have our class requirements well defined MainContainer is the only class we'll need to instruct Ext JS to load.

The last step to render our application on screen is to call Ext.setup and pass an onReady**{4}** method to it. Setting up this method will allow our application to launch at the correct time in the browser's page load cycle. We need this because our app will attempt to manipulate the DOM before it's fully built by the browser and without this, we'd get exceptions and never leave the launch pad.

Inside of the onReady function, we call Ext.create and pass String representation of our class name. With Sencha Touch Ext.create replaces use of the JavaScript new keyword, as it plays nicely with the Loader system.

New is so old!

I recognize that it may seem foreign to you at this stage in the game. It did to me at first as well. Just know that it will grow on you and if you peek at Chapter 9, where we dive into the class Loader system, you will gain an appreciation for Ext.create.

If we load our application in a Webkit Browser (I typically choose Google Chrome), we can see the class system at work.

Here's what I saw when I peeked at the Webkit debug tools.

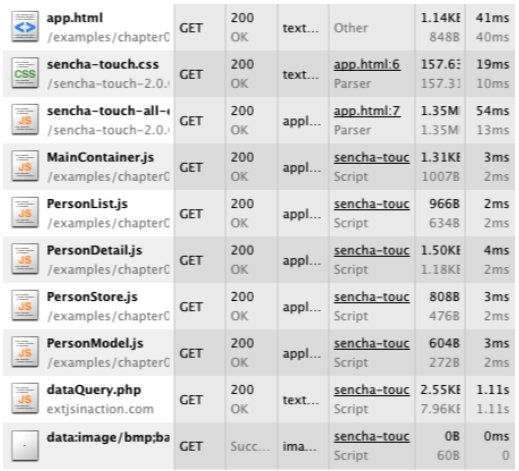

Figure 2.9 We can see the Sencha Touch Loader working as it loads our classes as defined in our classes.

Looking at the Webkit inspector "network" tab, I can see that immediately after Sencha Touch was loaded, our MainContainer class was loaded. Next, PersonList, then PersonDetail was loaded. The Loader system inspected the requirements for PersonList and loaded PersonStore, and loaded its requirement, PersonModel.

All of this sets the stage for our application to function. Here's what it looks like rendered in Chrome after I've selected the first record in the PeopleList UI.

Figure 2.10 Our application rendered inside of the Google Chrome

After rendering our miniature application, we can see that a selection of a record from the PeopleList widget (left) will result in the details of that record being displayed on the right.

This miniature read-only application sets the stage for having full-blown CRUD against the data, where we could create buttons for Create, Update and Delete operations, but I think this is a good time to stop with what we have for now as I hope it's wet your appetite a little for what is happening under the hood with this framework.

Understanding what critical concepts like lazy instantiation, which is what we'll be covering in the next chapter.

2.5 Summary

In this chapter, we covered quite a bit and learned a lot along the way.

We kicked things off by downloading the framework and taking a deep dive into its contents. Afterwards, we created a very simple Hello World example, where we implemented the MessageBox widget.

Next, we created a simple application. In this exercise you got exposure to critical bits of knowledge like defining classes, XTypes, class dependencies and event delegation. We rendered our application in Google Chrome and looked at how our classes were dynamically loaded and got a chance to exercise our application.

In the next chapter, we'll begin our journey into the heart of Sencha Touch, beginning with a deep dive into the Component lifecycle and discuss the various layouts in the framework. This will help us set the foundation for further exploration of this framework.

3

Sencha Touch Foundations

This chapter covers

- Getting to know the Component Model
- Explore the Component Lifecycle
- Learning utility methods to create `Components`

Sencha Touch `Components` are the building blocks of the Sencha Touch Universe. They are the foundation the rest of the framework relies on. Working with `Components` is a bit like working with Legos. No matter what you're building, you always stack blocks on top of each other in various ways. The wondrous things you create are only limited by your own imagination, and the rules of the framework of course. In order to understand just exactly how Sencha Touch enforces these rules, and how `Components` fit together, one has to understand the underlying Component Model and its relation to `Components`. During the course of this chapter, we will take a close look at the Component Lifecycle, and why you should care about it. In addition, we will take a look at the different ways to create `Components` and look at the pros and cons of each way. First things first, we need to build a good foundation to expand from, so let's take a look at the Component Model.

3.1 One Component Model to rule them all

Part of the magic behind Sencha Touch and what makes it so special compared to other frameworks is the way UI Widgets are structured, and that they fit together almost seamlessly. In every UI Widget, you can go far enough up the inheritance chain, and always arrive at the same common ancestor, the `Ext.AbstractComponent` class. Every UI piece in Sencha Touch is a subclass of `Ext.AbstractComponent`, something that is made possible via the Sencha Touch Component Model, a centralized model that governs all `Component`-related tasks. `Ext.AbstractComponent` is the base of the Component Model but should not be used directly, for this you should use `Ext.Component`. The Component

Model provides a set of rules and behaviors that all children of `Ext.AbstractComponent` have to follow in some way or another. Figure 3.1 depicts the common `Components` in the Component Model in all its glory, showing just how many UI pieces subclass `Component`, either directly or indirectly. Fret not if this figure seems overwhelming at first glance. It simply provides a big picture overview behind all of the Sencha Touch `Components`, and can serve as a good reference point to return to as you read later chapters, or venture into your own development endeavors.

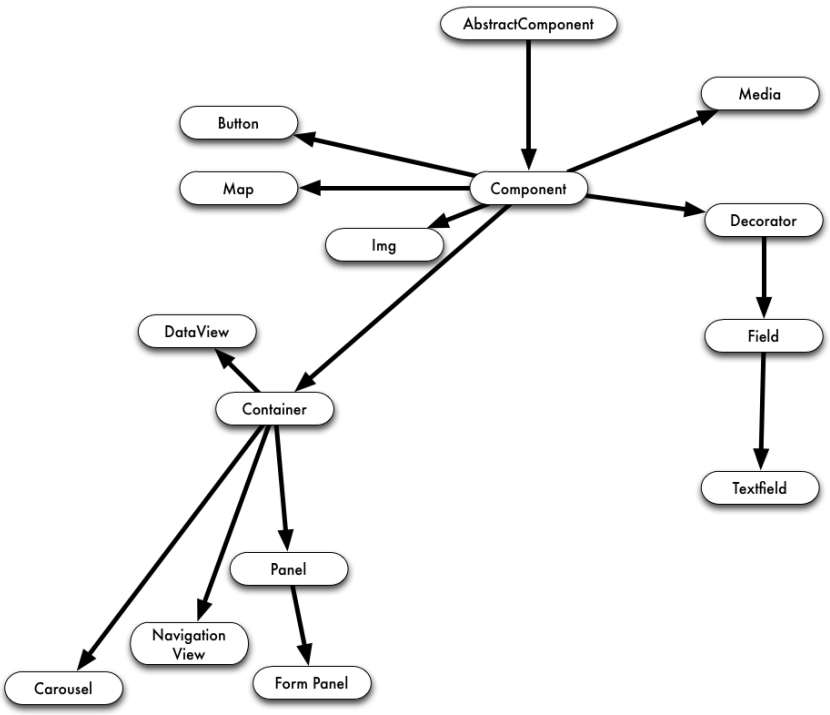

Figure 3.1 The common `Components` in the Sencha Touch Component Model depicting the hierarchy and subclassing.

Figure 3.1 shows the more common `Components` in Sencha Touch 2 that you will most likely use in everyday applications. These are not all the `Components` that Sencha Touch 2 has. Next in Figure 3.2 you will see the entire hierarchy of the Component Model in Sencha Touch 2, which may look overwhelming but is a great depiction of where one `Component` is in the hierarchy.

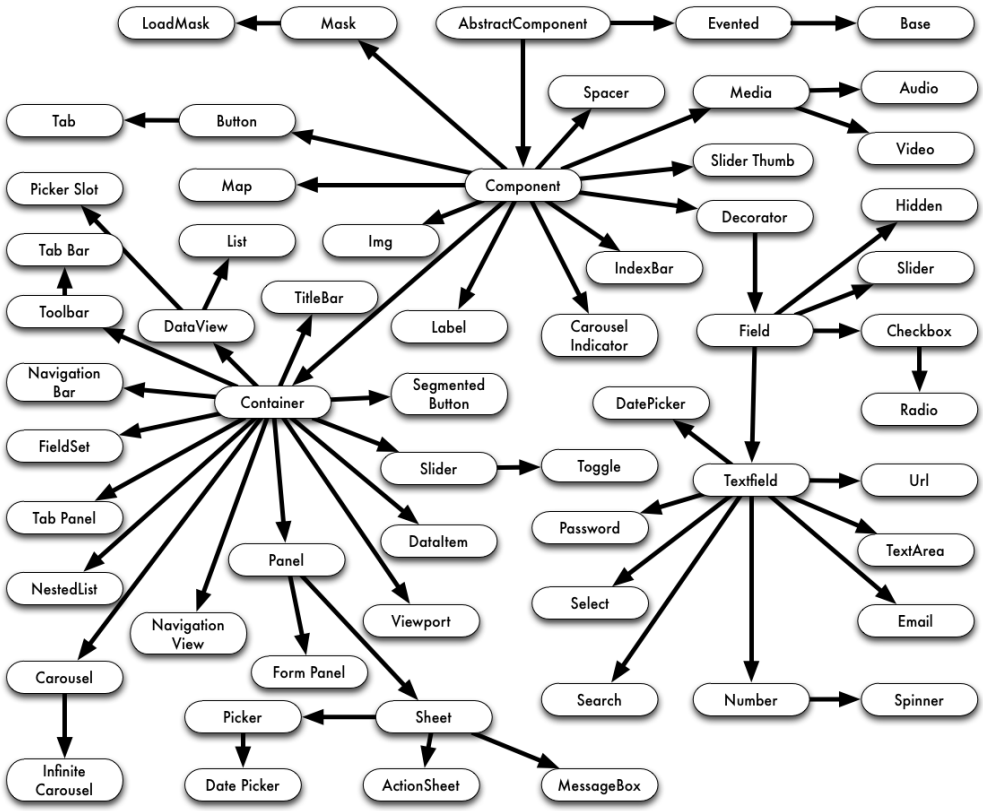

Figure 3.2 This illustration of the Sencha Touch class hierarchy maps out all the subclasses of `Ext.AbstractComponent`, and depicts how widely used the Component Model is in the framework.

One thing that might not be immediately apparently when looking at Figure 3.1 and Figure 3.2 is what exactly the benefit of the Component Model is.

Knowing what `Components` another `Component` extends is very valuable knowledge. The rules the Component Model has to adhere to allows for dependability. For example, you may know behavior that `Container` has and you can depend that it's subclasses, `Panel` for instance, will most likely behave in the same way; events will fire in a certain order. This dependability is one of the most overlooked benefits of the Component Model; whether you know it or not, you will be relying on this everyday you are developing with Sencha Touch.

Knowing exactly how each UI widget is going to behave introduces predictability and stability into the framework; No matter whether you are dealing with native Sencha Touch widgets, or `Components` created by 3rd party developers, you can always count on specific events and behaviors to occur at pre-determined points.

What makes all of this possible is the thing that governs how a component is instantiated, rendered, and destroyed, the Component Lifecycle.

3.2 Introducing the Component Lifecycle

Each component in the Sencha Touch Universe shares a common ancestor, `Ext.Component`. Just like in genetics, this ancestor passes down some of its traits. One of them being the Component Lifecycle, a set of rules and behaviors that determines what happens when a `Component` is "born", what happens while it "lives", and what happens when it "dies".

Given that the average app, created by the average user, frequently only consists of out-of-the-box `Components` without too much customization, the need to study all the gory details behind the Component Lifecycle can be assessed by each developer individually. For those "average" developers, the Component Lifecycle will be one of those things that they hear about on the forums, and that they will most likely have a high-level understanding of, which tells them that a `Component` is created during the Initialization Phase, rendered during the Render Phase, and destroyed during the Destruction Phase. Anything beyond that is not really a requirement – for the average app. For those developers who wish to eventually venture into creating their own custom `Components` and plugins that extend the core functionality of the framework, this somewhat dry topic will provide a goldmine of information. Quite a few steps take place at each of the three phases in the lifecycle.

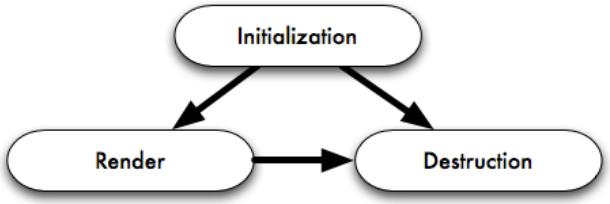

Figure 3.3 The Sencha Touch Component Lifecycle always starts with the Initialization Phase and always ends with the Destruction Phase. The `Component` does not need the Render Phase to be destroyed.

Since everything must have a beginning, we will start with the beginning of each `Component`, the Initialization Phase.

3.2.1 Initialization / Instantiation Phase

The Initialization Phase is when a `Component` is born. Any necessary pre-render actions like configuration settings & HTML structure creation take place during this phase. The exact steps break down as follows:

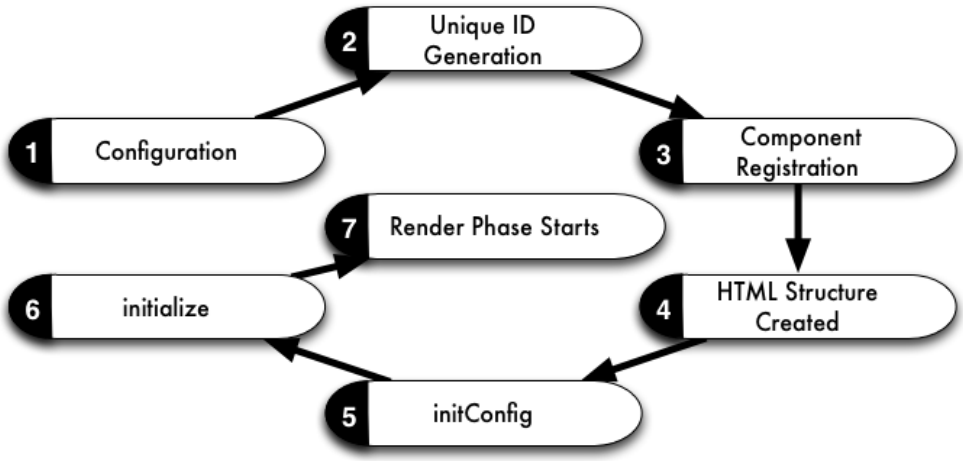

Figure 3.4 The Initialization Phase of the Component Lifecycle executes important steps such as `Component` registration and calling the `initConfig` and `initialize` methods. It's important to remember that a `Component` can be instantiated but may not be rendered.

Let's explore each step of the initialization phase. The numbers in Figure 3.1 correspond to each step in the list below.

1. *The `Component` configuration is applied* – When creating an instance of a component, you pass it a configuration object that contains all the necessary knobs and dials to tell the component what to do, and how to behave itself.
2. *Generate a unique ID* – Each `Component` within your app must have a unique identifier. This is a requirement dictated by the `Ext.ComponentManager`, since all `Components` are automatically registered with the `ComponentManager`.
   There are two ways a component can come by its ID.

   The first is for Sencha Touch to automatically generate an ID, which is the default and preferred behavior when no ID is supplied during `Component` instantiation. Sencha Touch generated ID's usually take the form of ext-component-1, ext-component-2, and so forth. The format is ext-{xtype}-{number} where {xtype} is the XType of the `Component` and {number} is an incremendtal number.

   The second way is for you (the developer) to supply an ID during instantiation, thus providing more control over what the ID should be. The ID can be any string you desire, the only caveat to keep in mind is that the `ComponentManager` does not allow duplicate ID's within the entire application. This means it is up to you keep track of custom ID's you set and ensure there are no duplicates. Whenever a `Component` with an already existing ID is registered, the `ComponentManager` overwrites the

already registered version with the newly supplied one. Nuking existing components causes very undesirable effects and is a pain to debug, so be careful about this.

ItemId to the rescue

One easy way to alleviate having to track countless ID's across your entire application is to use the `itemId` property for ID's assigned by you, and to let Sencha Touch automatically assign the `id` whenever possible. The `itemId` simply has to be unique within a container, meaning you could have two `Panel` instances, each containing a `Textfield` with an `itemId` of "myTextField" and not run into any collision issues.

ComponentQuery

A more flexible and robust way is to make use of `Ext.ComponentQuery` taking advantage of the hierarchy of your application. You can go up and down the hierarchy searching XTypes and properties. The caveat is there is a small performance hit when using this but in today's devices that are getting more powerful every day, this is quickly loosing concern.

`Component` is registered with `ComponentManager` – Each instance of a `Component` is automatically registered with the `Ext.ComponentManagerr` using its unique ID. If you know this unique ID, you can utilize the `Ext.ComponentManager` to perform a lookup of the component and get a handle on it for interaction via the `Ext.getCmp` function.

3. *HTML structure is created* – Each Component has it's own HTML structure for it to display. This step creates the HTML structure but is not placed in the DOM to be rendered yet. Sencha Touch 2 creates this structure so it can do one batch DOM write instead of doing multiple DOM write. This helps with performance as one of the slowest things that will happen in your app is DOM write so minimizing the number of writes is a very good way of improving performance. Remember, we are still in the Initialization Phase; the Component won't be actually rendered until the Render phase later in the Component Lifecycle.
4. *`initConfig` is executed* – One of the new key features in Sencha Touch 2 is the `config` object. This `config` object automatically creates a getter and setter method for each `config` object item. `initConfig` is the method that every `Component` (and actually every class) has and does the creating of the getter and setter methods. A getter method will simply return the property value that was in the `config` object. A setter method will not only set the property value but also uses an apply and update method. When you execute the created setter method, if first looks for an apply method and executes it passing in the new value and old value as arguments. The apply method is great for transforming the value or providing validation and it must

return a value. After the apply method is executed the property value is then set. Then the update method is fired passing the new value and old value. The update method is great to take action on the setting of a new value like updating some other Component or DOM element.

5. *initialize is executed* – The initialize method is used to add additional configuration before the Render Phase starts. This is the last template method that you can override for subclassing before the Component is rendered. This can provide a last chance to change items or apply event listeners. Do note that some Sencha Touch Components use the initialize so if you need to override this method, you will usually have to execute callParent to call the superclass' method.
6. *Component is rendered* – If the fullscreen config is set to true the Component will fire the fullscreen event which the global Ext.Viewport Component listens to and will automatically add the Component to itself triggering the rendering.

This phase of a Component's life is usually the swiftest, due to the fact that all of the work is done behind the scenes in JavaScript, and no rendering is necessary up to this point. It is particularly important to remember that a Component does not have to be rendered in order for it to be destroyed.

3.2.2 Render Phase

The Render Phase provides visual feedback that a Component has actually been successfully instantiated. If the Component encountered any issues during the Initialization Phase, it is unlikely to render correctly or at all. Unless the fullscreen property is specified, rendering of the Component is deferred until the Component is added as an item of a Container. We will talk more about Containers in the next chapter.

If your Component is a child of a Container, the Container's layout usually takes care of doing the rendering. The following code shows how to use the fullscreen property to have Ext.Viewport automatically add the Component as an item.

```
var myComponent = new Ext.Component({
    fullscreen : true,
    html       : 'testing'
});
```

In this example, you instantiate an instance of Ext.Component and create a reference to it, myComponent. After the instance is instantiated, it sees the fullscreen property is set to true and will then fire the fullscreen event on itself. Ext.Viewport listens for the fullscreen event and when a Component fires this, it will automatically add that Component as an item of itself.

In Sencha Touch 1 you usually used the renderTo or fullscreen config, which would render that Component to a particular element, or you could add a Component to a Container and that would then render the Component within the Container. In Sencha Touch

2 you can still render Components to an element but you are going to add Components as items of Containers 99% of the time.

Also in Sencha Touch 1 the layouts were JavaScript based which didn't perform well and wasn't 100% accurate. In Sencha Touch 2 layouts are CSS based which is going to be highly accurate and performs very well. In the case of an orientation change, the layouts are usually finished before the browser has finished updating it's size so you can't get much faster than that.

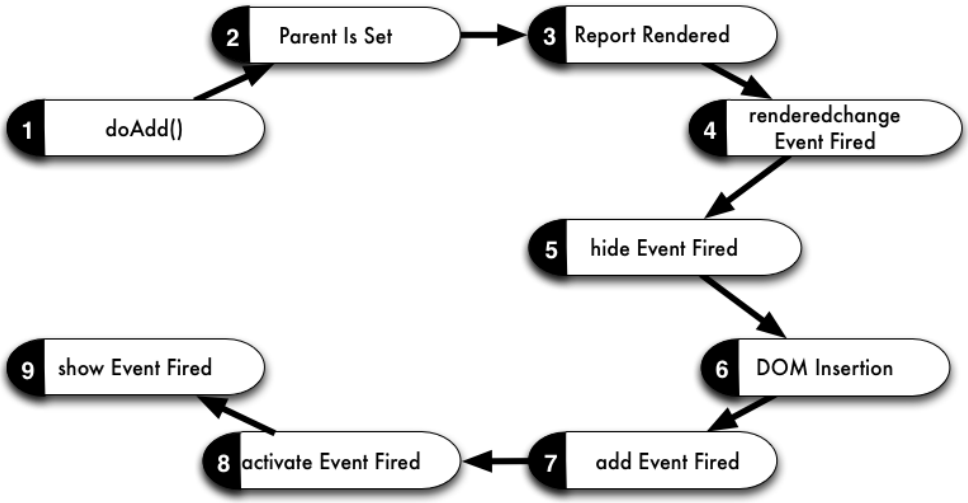

Figure 3.5 The Render Phase of a `Component` is pretty simple as layouts are CSS based.

The actual `Component` rendering is split into multiple steps as can be seen in Figure 3.4 above. The number for each step corresponds to the number in the list below. There are a few steps that are big steps but are important. Let's explore the different steps of the Render Phase:

1. *doAdd is called* – The `doAdd` method is called adding the `Component` to the `Container` as an item. This doesn't really do rendering but kicks off the rendering.
2. *`setParent` is called* – The `Component` needs to have a defined parent. This method first checks to see if the parent property on the `Component` is set and if it is then it will automatically remove the `Component` from the current parent. Then it will simply set the parent property to the `Container` the `Component` is being added to.
3. *`Component` reported rendered* – At this point the `Component` reports that it is rendered. This is a little tricky, as it isn't rendered in the sense that it's in the DOM. The HTML structure was already created back in the Initialization Phase so it's

rendered outside of the DOM. A little crazy to think that way but it gives the framework a chance to change the HTML before it's actually inserted into the DOM. Remember from earlier that DOM writing is slow, doing it this way saves a few crucial milliseconds.

4. *renderedchange event is fired* – Since the Component now reports it is rendered, the renderedchange event is now fired. If you use this event to try and do custom things to the element of the Component, you need to be aware that the Component isn't in the DOM at this point. This is actually a very important thing to know, you can change do things to the element but it may be limited as it's not in the DOM.
5. *hide event is fired* – This is another important step. The Component is still not in the DOM but it is getting hidden. But why? When the Component is finally inserted into the DOM the Component is hidden in order for the layouts to correctly catch up so you don't see the browser jerking things around. This gives a much better user experience. It's like the quote, "Sometimes you have to take a step back to go forward." You will see in the following steps that there are still things that are going to happen on the Component.
6. *DOM inserted* – It's been a long road but yes, the Component is now finally inserted into the DOM! The Component is still hidden but it's there. A few more things to do and it will be made visible.
7. *add event is fired* – The parent of the Component will now fire an add event. This isn't really a step for the Component as it's parent is the one that fires the add event but it's part of the rendering process of the Component and could be important to know about as you develop your application.
8. *activate event is fired* – This event will only fire if the Container of the Component doesn't already have an active item. If the Component is to be made the active item then it will have the activate event fired. This will also trigger the Component to be made visually active so it is no longer hidden but do remember the event is fired before the Component is visually shown. The show event is fired after the Component is visually shown at this point.

The Render Phase is made up of a lot of events. Being event driven allows code to remain flexible. The same thing could be done by executing methods but the same events the framework uses to render a Component can also be listened to when you are developing an application; methods are much harder to hook into. So we have studied how a Component is created in the Initialization Phase and optionally rendered in the Render Phase, the last part of the Component Lifecycle is to destroy the Component in the Destruction Phase.

3.2.3 Destruction Phase

The death of a `Component` is a crucial step in its life. Destructing the `Component` is performed via the `destroy` method, which takes care of critical tasks such as removing the `Component` and any child from the DOM tree, purging event listeners, as well as unregistering the `Component` from the `Ext.ComponentManager`. The `Component`'s `destroy` method could be called by a parent `Container` or manually by your code. Figure 3.6 illustrates all the steps that go into the Destruction Phase.

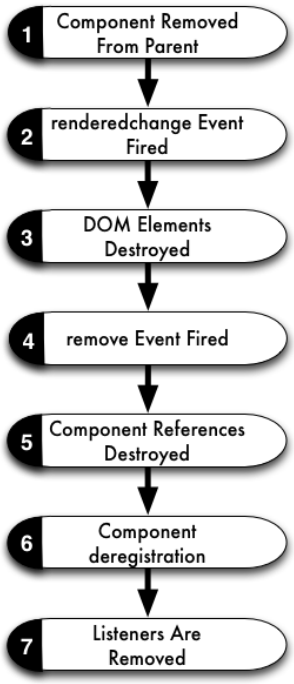

Figure 3.6 The Destruction Phase of a `Component` is equally as important as the Initialization Phase. This is where cleanup of event listeners and DOM elements, as well as deregistration and removing of Elements happens. All in the spirit of keeping your application running like the well oiled machine that it is.

Just like it was with our Lego blocks from childhood, it was always harder to build something than to tear it down. The Destruction Phase is no different in that regard. Here are the steps that make up the final stage of a `Component`'s life. Keep in mind that each step below corresponds to a step in Figure 3.6.

1. *Parent removal* – If the `Component` is an item of a `Container`, the `Component` will be removed from the `Container`.

2. *renderedchange event is fired* – The renderedchange event is fired telling the Component that it will no longer be rendered. It is good to note that the event is fired before the Component is actually removed.
3. *DOM removal* – Unlike the Render Phase where we had to wait for the DOM to reflect the Component, the destruction of the Component removes the DOM elements early on.
4. *remove event is fired* – Now is the time for the Container to announce that the Component has been removed by firing the remove event. The remove event is an important step in the Destruction Phase even though the event is not fired on the Component itself.
5. *References are destroyed* – Each Component has a referenceList property that is an Array of Strings with each string being a property on the Component. For example, all Components will have the string 'element' in the referenceList Array and every Component has an element property which is an Ext.dom.Element of the Component's DOM. This step loops through each item and executes the destroy method on each item. Even though the DOM elements for the Component, there is still an Ext.dom.Element reference to it and if it wasn't destroyed then this would be a memory leak.

   An advanced technic when you are creating custom Components is you can utilize this referenceList Array to clean up references, you just have to make sure it has a destroy method or else you will get a JavaScript error.
6. *Component deregistration* – The reference for the Component in the Ext.ComponentManager is removed.
7. *Listeners are purged* – All listeners on the Component are cleared and removed from the Component. If this step did not happen, the leftover listeners would be considered a memory leak and would quickly add up and slow down your application.

Now that you've endured the long and arduous road through the Component Lifecycle, it is important not to underestimate the importance of the three different phases. Especially when venturing into creating your own components. Many developers have dismissed the Destruction Phase, only to leave a continuously polling data Store hanging around to cause havoc.

The last part of this chapter is about how Components can actually be created and how they are managed once they are created.

3.3 XTypes and the ComponentManager

Each Component within the Sencha Touch framework allows for three ways to be created: Direct Instantiation, Direct-Loaded Instantiation and Lazy Instantiation. So what's the difference? Direct Instantiation will instantiate that Component right away when the browser executes that code. Direct-Loaded Instantiation is very similar to Direct Instantiation that the Component will be created right away but only after checking if the class is loaded.

Sencha Touch can dynamically load Components so in order to create a Component, the file has to be loaded and then the Component can be instantiated. Lazy Instantiation is where you use a configuration object. Once the Component is required to be instantiated the Component will then be instantiated using that configuration object.

3.3.1 Examples of Instantiations

Direct Instantiation will look familiar to most programmers as most languages primarily use it. In this case the Component is created explicitly via its constructor, providing instant access to the Component and all of its properties, including the Component's DOM structure once it renders.

In practice, it looks something like this:

```
var myPanel = new Ext.Panel({
    modal            : true,
    hideOnMaskTap    : true,
    centered         : true,
    scrollable       : 'vertical',
    width            : 300,
    height           : 200,
    styleHtmlContent : true,
    html             : '<h2>Hello Readers</h2>'
});
```

The Direct-Loaded Instantiation method makes use of Ext.create. Ext.create can create a Sencha Touch Component. This is very similar Direct Instantiation method where it will create the Component immediately but uses Ext.Loader. Using Ext.Loader if the Component isn't loaded then Ext.Loader will load that class and then create the Component like Direct Instantiation via the constructor. We will take a look at Ext.Loader later in this chapter. An example of using Ext.create is like this:

```
var myPanel = Ext.create('Ext.Panel', {
    modal            : true,
    hideOnMaskTap    : true,
    centered         : true,
    scrollable       : 'vertical',
    width            : 300,
    height           : 200,
    styleHtmlContent : true,
    html             : '<h2>Hello Readers</h2>'
});
```

Sencha Touch allows for the use of Lazy Instantiation, an alternative method to Direct and Direct-Loaded Instantiation that uses an XType to represent the Component.

```
var myPanel = {
    xtype            : 'panel',
    modal            : true,
    hideOnMaskTap    : true,
```

```
        centered         : true,
        scrollable       : 'vertical',
        width            : 300,
        height           : 200,
        styleHtmlContent : true,
        html             : '<h2>Hello Readers</h2>'

    };

    Ext.ComponentManager.create(myPanel).show();
```

Instead of explicitly creating a new instance of a `Component`, a plain JavaScript Object containing a special String property named an XType along with the other various configuration values for the class is used in its place. The XType value serves as a unique identifier linking the string value to an actual Sencha Touch `Component` registered with the `Ext.ComponentManager`, a singleton class that tracks all `Components` and their XType. This link tells the `ComponentManager` which `Component` to instantiate when an XType is encountered. The reason the code sample above works is because every single UI Widget is registered with the `Ext.ComponentManager`, and the code above uses a `'panel'` as the XType, which corresponds to the `Ext.Panel` `Component`.

Whenever a `Component` is initialized, the `ComponentManager` automatically checks whether the `Component` contains any child items, and for each child item checks whether the XType property is set. If that is the case, a new instance of the class that corresponds to the XType is created where the child `Component` should be.

3.3.2 The pros and cons

Now that we know the ways to instantiate a `Component`, when should we use one over the other? We first need to look at some pros and cons of using each method in order to make a decision what to use.

Direct Instantiation will create the `Component` right away, which is good right? Yes and no. It is good because that `Component` and all the properties and methods are available right away but the issue with this is in some situations you do not need to have that `Component` instantiated right away. Direct-Loaded Instantiation has the same issue but could make an AJAX request to load the `Component`'s source file which could delay start-up time and therefore giving the illusion to the user using the application that the application is running slow and performance takes a negative hit.

If that `Component` isn't being used then it may not actually need to be instantiated and taking up precious memory on the mobile device; this is where Lazy Instantiation may be a better choice. When you do not need to have all those properties and methods available, you can use a Lazy Instantiation to have only a small object be stored in memory. Once the `Component` is needed then it will be instantiated into a full `Component` thus saving memory on that mobile device and your application will perform better. Later in Chapter 9 you will learn about creating your own class using Ext.define and in the `config` property you will be using Lazy Instantiation.

Another benefit of using Lazy Instantiation is your code will be cleaner and more elegant. Child items can be declared inline versus creating them one-by-one, resulting in more streamlined code. This configuration object can be assembled dynamically by your code and even be retrieved from a backend via AJAX.

So Lazy Instantiation sounds fantastic doesn't it? Sometimes, when something sounds too good to be true, in some instances it can be. Using Lazy Instantiation defers both the Instantiation Phase and Render Phase of the Component Lifecycle resulting in a loss of immediate access to the `Component`, something that is not a problem when using Direct or Direct-Loaded Instantiation.

If you need to take action on a `Component`, it must be instantiated. If the `Component` hasn't been transformed into a full `Component`, you can't really do anything with it. You always need to be aware of this limitation in your application, is that `Component` going to be a `Component` or is it going to be a configuration object?

There are many things to take into consideration but don't be overwhelmed. The decision between Direct Instantiation and Direct-Loaded Instantiation is very simple, could that `Component`'s source file not be loaded? If it may not then Direct-Loaded Instantiation is the way to go; if it is going to be loaded then using Ext.create may cause a little overhead that isn't needed and you can just go with Direct Instantiation. The decision between Direct/Direct-Loaded Instantiation and Lazy Instantiation can be simplified down to whether or not that `Component` is needed when the lines of code is executed by the browser. Lazy Instantiation can allow you to boost performance but it can also create more work by dealing with the fact it's not a `Component`. If you can get away with using Lazy Instantiation, it is recommended over the other two.

3.4 Summary

This chapter may not be the most glamorous chapter but is a very important chapter. Before you can build walls of a house you must have a stable foundation. We saw the entire Component Model, checked out the Component Lifecycle and it's three phases and lastly explored how to create a `Component` and some things to think about when deciding on how to create a `Component` in your application. I hope the importance of this chapter was well received but we have only scratched the surface of what Sencha Touch provides and what is possible from this. If you aren't yet comfortable with the material covered in this chapter if might be prudent to move on and periodically circle back to this chapter as a refresher. Ready for some more excitement, next we are going to take a look at `Containers` and some different `Panels` to give Sencha Touch some more life.

4

Mastering the building blocks

This chapter covers

- Exploring the Sencha Touch Container model
- Learning utility methods to arrange and manage widgets
- Learning to float and drag Panels
- Implementing the TabPanel

In the last chapter, we covered a lot of foundational topics, including the Component Model and Component Lifecycle, and most importantly, how to create Components. Armed with all this knowledge, we can now show off how Sencha Touch leverages these new concepts in some of the major UI components, and thus set the stage for you to master such topics as Containers, Panels, and Tab Panels; exactly the things covered in this chapter. We will explore how components are managed within Containers, and how to arrange components in various ways using the provided Layouts. In the process, we will take a gander at building a simple wizard-style interface, and explore how to build a Tab Panel that looks similar to the typical iOS apps with icons at the bottom.

First things first, we'll begin by taking a deep dive into the anatomy of the Container class.

4.1 Containers: Mounting our UI Workhorse

The `Container` is the workhorse widget of Sencha Touch, as it is typically deployed to display and help organize your application layout. It provides the foundation for `Components` to manage their unruly children, and facilitates ways to add, insert, remove, and query child components. `Container` is an extremely versatile widget and is even deployed by the framework authors to add other UI components to the framework. In order to feel the gravity of the container's importance, take a look at its class inheritance model below.

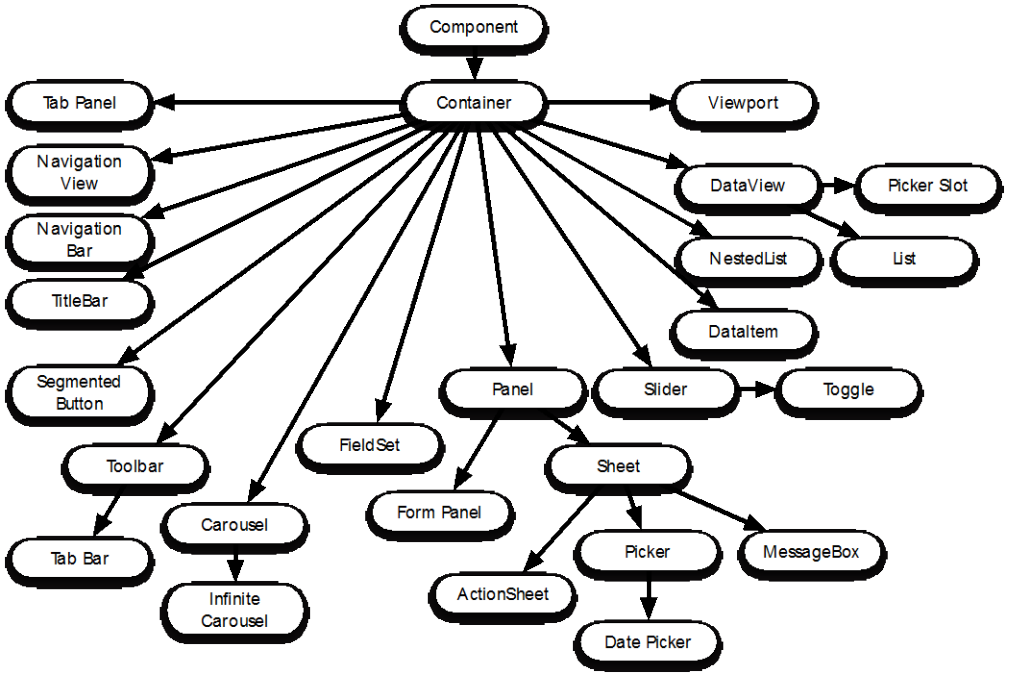

Figure 4.1 Container Hierarchy

As illustrated in Figure 4.1, `Container` is the base for at least 14 immediate subclasses, including widgets that you might not expect like `NestedList`, `Carousel` and `Toolbar`.

In case you're wondering, the reason why `Container` is used as the base for these widgets will become much clearer when we take a look under the hood of this widget.

4.1.1 Container's anatomy

The Container classes' versatility stems from the ability to contain child widgets, and the ability to easily manage and arrange them. The container leverages a layout manager to visually organize child items in an area known as the content body, or "body" for short. To understand what that looks like, let's go ahead and build our first container.

Listing 4.1 Building our first Container

```
var loginContainer = new Ext.Container({
     id: " myLoginContainer ",
     fullscreen: true,
     items: [
        {
```

```
                xtype: "textfield",
                label: "Login",
                placeHolder: "Enter Username Here"
            },{
                xtype: "textfield",
                label: "Organization",
                itemId: "orgField",
                placeHolder: "Enter Your Organization Here"
            },{
                xtype: "textfield",
                label: "Password",
                placeHolder: "Enter Password Here"
            }
        ]
});
```

The code itself is relatively simple. We instantiate a `Container` and give it an `id`. Since we will be reusing this sample code for later parts of this section, this is necessary so we can obtain a handle on the container. Since this is our one and only container on the screen, we mark it as `fullscreen:true`. This ensures that the container automatically utilizes the full height and width of the device it is run on. After that, we simply add a few form elements via `xtype`. Once the code executes, it should look like figure 4.2 below.

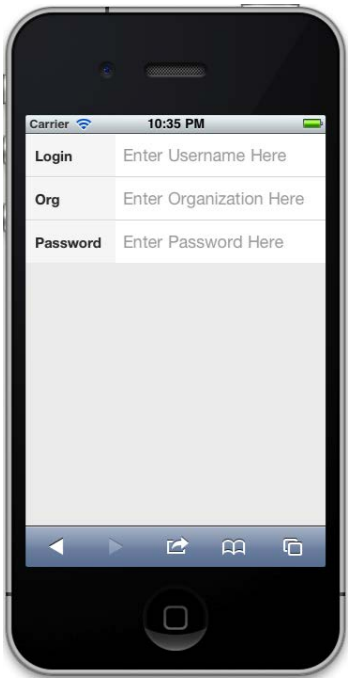

Figure 4.2 The rendered Container UI for listing 4.1

We have our first `Container`, with a Login `Form` nonetheless. Obviously, we will want to interact with the `Components` for various reasons like obtaining values from the `Form` elements, or because we need to change the content of the `Container` to fit our particular application logic. In order to do this, we will need to learn how to obtain references to the `Container` and its children, and how to make changes to `Container`.

4.1.2 Keeping unruly children on the right track

Dealing with unruly children is a frustrating fact of life for any parent. Over the years you develop tricks and ways to help yourself in that endeavor. Things like timeouts, revoking TV privileges, or simply handing them off to your parents for a while.

Just like real life, we must develop ways to handle children within containers to keep them in-line. Luckily for you, most of the helpers & utility methods you need to accomplish this are already in place. Mastering these methods will enable you to dynamically update the UI of your app and make it more interactive. The only thing left is getting acquainted with these methods.

Adding components into a container is a relatively simple task and you are provided with two methods: `add` and `insert`. The `add` method simply appends the new child `Component` to the `Container`'s hierarchy, while the `insert` method places the new `Component` at the specified index. To illustrate the `add` functionality further; let's add to the Container created in listing 4.1. For this, we'll use our handy Safari Debug Console:

```
Ext.getCmp("myLoginContainer ").add({
     xtype: "checkboxfield",
     label: "Remember"
});
```

Running the preceding code will add a new `Component` (a checkbox) to the `Container`. The container's layout automatically handles refreshing the container and updating positioning and sizing of all child components to make the new component appear in the right place.

EXTJS AND SENCHA TOUCH 1.X DEVELOPERS TAKE NOTE

For those developers upgrading from Sencha Touch 1.x (or coming from ExtJS), you might have noticed that the new component shows up immediately. This is due to the new layout manager, which removes the need to call `doLayout()` separately.

Figure 4.3 Illustration of code from listing 4.1 with a checkbox added dynamically. Notice that the checkbox was appended at the end.

Inserting a `Component` works basically the same as adding, the main difference being the additional parameter to determine the index where the new `Component` should be inserted. In this example we are inserting a new item at index 0, making the new `Component` the first item in the `Container`.

```
Ext.getCmp("myLoginContainer ").insert(0, {
     html: "<h1>Please enter your credentials</h1>"
});
```

Running this snippet of code from the Safari Debug Console will result in figure 4.4.

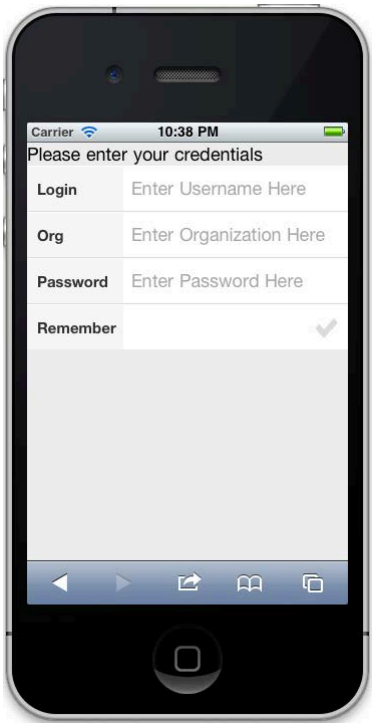

Figure 4.4 Rendered UI using code from listing 4.1 with dynamically added item shown in Figure 4.3 as well as an inserted panel at position 0. Since the panel doesn't have any borders around it, it simply shows as an html snippet at the top.

As you can see, adding and inserting components is easy as pie. Removing components is no different in that regard. You either provide a reference to the `Component` that should be removed from the `Container`, or the `Component`'s `id` or `itemId`.

```
Ext.getCmp("myLoginContainer ").remove("orgField");
```

Executing the statement from the Debugger console causes the component to invoke the `Components destroy` method automatically, thus initiating the destruction phase and deleting the `Component`'s DOM element.

One special thing to note about the `remove` method is an optional second parameter that can be passed to tell the removed `Component` to skip invoking the `destroy` method. This is particularly useful if you wish to move a `Component` from one `Container` to another. In principle this works by obtaining a reference to the `Component` you wish to remove and then removing the `Component` while passing "false" as the additional second parameter. This leaves the `Component` on the screen and only removes it from the

Container. You then add the component (using the reference you still have) to the second Container, thus triggering the DOM elements for the removed Component being moved into the new Container. To translate this into code, take a peek at listing 4.2

Listing 4.2 Moving Items between Containers

```
var mainContainer = new Ext.Container({
    fullscreen: true,
     defaults: {
            style: "margin-bottom:30px"                                    1
     },
     items: [
        {
             itemId: "panel1",
             items: [
                {
                     xtype: "textfield",
                     label: "Login",
                     itemId: "loginField",
                     placeHolder: "Enter Username Here"
                }, {
                     xtype: "textfield",
                     label: "Password",
                     placeHolder: "Enter Password Here"
                }
             ]
        }, {
             itemId: "panel2",
             items: [
                {
                     xtype: "textfield",
                     label: "Organization",
                     placeHolder: "Enter Your Organization Here"
                }, {
                     xtype: "checkboxfield",
                     label: "Remember Login"
                }
             ]
        }
     ]
});


Ext.defer(function(){                                                      2
     var myField = mainContainer.down("#loginField");

     var panel1 = mainContainer.down("#panel1");                             3
     panel1.remove(myField, false);

     var panel2 = mainContainer.down("#panel2");
     panel2.add(myField);
}, 2000, this);
```

1. **Define defaults values for children**
2. **Defer function execution by 2 seconds**

3. **Obtain a handle on children**

Listing 4.2 might look complicated at first, but it breaks down relatively simple. We are creating an `Ext.Container` that contains two child panels, each of them with a different `itemId` and two form elements, `textfield`s and `checkbox`es respectively. The main panel contains a `defaults` property **(1)** that contains configuration options that are automatically applied to all direct children of it, in this case `panel1` and `panel2`.

After instantiation, we create a function **(2)** whose execution is deferred by 2 seconds. For this, we are using the `Ext.defer` method, which is similar to the standard JavaScript `setTimeout` function. It defers the execution of a function by a specified amount of milliseconds. What makes it special is that it allows you to set the scope of the function execution, thus allowing you to run the deferred function in the right context with the variables you need available. The deferred function first obtains a reference on the field we wish to move, the `loginField`, and then a reference to the panel we wish to remove it from **(3)**, `panel1`. We call `panel1.remove` and pass in the reference to the `loginfield` and false, to indicate that we want to keep the `loginField` around after it is removed from `panel1`. We then simply obtain a handle on `panel2` and add the `loginField` there.

At this point, you are probably wondering exactly how it is that we obtained a reference on the items we wanted, and how the `down` method works. For this and more, stay tuned for the next part of the chapter.

4.1.3 Ask and ye shall receive – Querying the Container hierarchy

`Containers` provide multiple ways for you to find and get a handle on children. Generally, these methods can be split into two groups. The first being methods that only return **direct children** of the `Container`, the second being methods that return children at any level. To illustrate the difference, imagine an `Ext.Container` that contains an `Ext.Fieldset` with a single `textfield`. The first set of methods would only be able to reach the fieldset, since that is a direct child of the panel. The second set of methods would be able to return the text field, since it can find items at any level, including searching within items of the Container.

Out of all these methods, `getComponent` is by far the easiest, and falls into the first group. It only accepts a single parameter, which can either be an index or a string. The index corresponds to the index location of the item within the `items` collections of the `Container`. The string corresponds either to the `id` or the `itemId` of the `Component` you wish to retrieve. In practice, using `getComponent` looks like this:

```
mainContainer.getComponent(0);

mainContainer.getComponent("panel2");
```

Not much to it, eh? The first line retrieves the very first child `Component` within `myPanel`, and the second line retrieves a `Component` with an `id` or `itemId` of “panel2”. If we assume a code structure like listing 4.2 above, then the first line would retrieve panel1 and the

second line would retrieve panel2. Keep in mind that this method only returns direct children of the `Container` you are querying.

While using `getComponent` is relatively simple, it is somewhat limited. This is why Sencha Touch presents us with three functions that have more flexibility in what and how you query.

All three of these functions take a single selector string that behaves similar to how CSS selectors work, allowing you to query for items either by id, itemId, XType, or any combination thereof. To draw more parallels to CSS, you can think of id and itemId the same as when you're addressing an id in CSS, via the pound sign. Likewise XTypes are similar to classes, requiring the use of a dot to address them.

The first of these functions is the `child` method, which falls into the first group, returning only direct children of a `Container`. We will once again use the code structure from Listing 4.2 above as a baseline for our next example, consisting of multiple calls to the `child` method. Each call will pass in one of the selectors from the following table, just like this:

```
mainContainer.child("SELECTOR-GOES-HERE");
```

Table 4.1 These sample selectors should help you differentiate between the different selectors the child, down, and query methods accept.

Selector	Explanation & Result
`"#panel1"`	The pound sign indicates that we are searching for an item by `id` or `itemId`, in this case "panel1". Since `panel1` is first item that is a direct child of `mainPanel` and matches the `itemId,` it is returned.
`"#loginField"`	Here we are searching for the loginField by `itemId`. Since `loginField` is not a direct child of `mainPanel`, null is returned
`".panel"`	This line searches for items with the panel XType. This is indicated by the dot in the beginning. This would return `panel1` since it is the first direct child with an XType of panel.
`".textfield"`	This line would once again return null, since we are searching for items with XType textfield, of which none are direct children of `mainContainer`.

Looking at the table, you can immediately see the benefits of using `child` to make your queries. As mentioned before, it is not the only method. The other two are the `down` and `query` method. Both follow the exact same syntax as the `child` method, with the only distinction that all sub-levels of a `Container` are searched, not just direct children.
If we were to rewrite the second sample from the table above to use `down` instead of `child` it would look like the following, and return the login field.

```
mainContainer.down("#loginField");
```

Both `down` and `child` only return a single item, more specifically they return the first match encountered. If you anticipate having multiple results and want all of them, use the `query` method. Rewriting the last sample call from the table to use `query` returns an array of 3 items, all of them text fields.

```
mainContainer.query(".textfield");
```

The only thing to keep in mind about `query` that makes it slightly different from the other methods is that it always returns an array, even if a single item or nothing was found the array would simply contain a single item or be empty.

A note about good form

When using the new component query functions to search by xtype, a dot prefix (.xtype) is optional, it is considered good form to use it anyway, since it makes it very clear that you are searching for an xtype.

We have covered `Containers` and how to fill them with `Components`, as well as how to add, edit, and remove `Components` from `Containers`. Each example so far has only dealt with the most basic way to arrange `Components` by stacking them on top of each other, using the `default` layout. Not sure about you, but I want my applications to be more visually appealing than simply having elements stacked one above the other. For that, we are going to take a closer look at the magic behind arranging `Components` within your `Containers`, the Sencha Layouts.

4.2 Everything must have its place – Layouts

One of the hardest parts when building any app is managing all the UI pieces. Positioning everything correctly so it shows up where it is supposed to, and making sure things remain that way after the user starts wildly thrashing around in your beautifully laid out application are difficult to achieve. This is where the Sencha Touch layout management schemes come to the rescue. They are responsible for managing the visual organization of all components on-screen. The complexity of these layouts range from a simple `FitLayout`, which

automatically sizes a single child item to fit within the confines of its parent, to more complex layouts like the `CardLayout`, which provides a wizard-like interface with multiple screens.

When exploring these layouts, we will hit upon some more lengthy and verbose examples that can serve as a great starting point for your application. We will start our journey by taking a look at the building block for all layouts, the `default` layout.

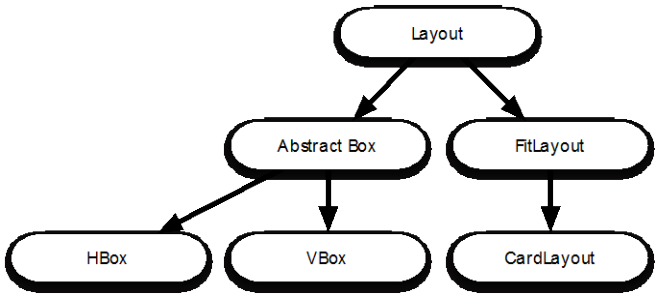

Figure 4.5 The layout class hierarchy, where all layouts subclass the default layout.

4.2.1 The Default Layout

This layout serves as the foundation for all other layouts. Besides managing items that need to be laid out and positioning them, it also provides the capability to dock items. This ability is in turn passed down to all other layouts, and will be covered in more detail in the next chapter.

The default layout is the easiest to utilize since it is automatically assigned for any `Container` that doesn't have a specific layout defined for it. Any items within a container utilizing this layout are simply placed on the screen, one on top of another. Items are not directly size managed, meaning that any item can conform to the size of the parent, but doesn't necessarily have to do so. In order to see this, you need to setup a dynamic example using a few components as shown here:

Listing 4.3 Default Layout

```
var myContainer = new Ext.Container({
     fullscreen: true,                                          1
     defaults: {
            style: 'border: 1px solid blue;'                    2
     },
     items:[
        {
            docked : 'top',
            xtype: 'toolbar',
            title: 'Default Layout',                            3
        },{
            docked : 'bottom',
```

```
            xtype: 'toolbar',                                          4
            items: [
               {
                    text: 'Add Child',
                    handler: function(){
                        myContainer.add({
                            xtype: ' container ',
                            style: 'border: 1px solid blue;',
                            html: 'Child'
                        });
                    }
               },{
                    text: 'Add Fixed Width Child',
                    handler: function(){
                        myContainer.add({
                            xtype: 'container',
                            style: 'border: 1px solid blue;',
                            width: 100,
                            html: 'Fixed Child'
                        });
                    }
               }
            ]
        },{
            html: 'First Child'                                        5
        },{
            html: 'Fixed Width Child',
            width: 100
        },{
            html: 'Child',
        }
    ]
});
```

1. **Container should occupy the full screen**
2. **Setting default values for all children (excluding docked items)**
3. **Creating a title bar**
4. **Creating a bottom docked toolbar**
5. **Adding child panels**

In listing 4.3, we do quite a lot to exercise the `DefaultLayout`. The primary reason here is to illustrate how items stack and don't resize when added dynamically.

The first thing to do is instantiate a `Container` that occupies the entire screen through the `fullscreen` option **(1)**. Since we will be adding simple child containers, which don't come with any visual indicators to mark their bounding boxes, we will once again use the `defaults` object **(2)** to give each child automatically a blue one-pixel border. To be a bit more user friendly, we add a title bar **(3)**, which is simply a toolbar with `docked:'top'` and a `title` property. After that, we add a second toolbar at the bottom, with two buttons **(4)**, to allow adding of new items that have a fixed width and items that don't have a fixed width. Note that although the toolbars are defined inside of the `items` array, they contain the `docked` option, which means they don't show as part of the container body itself. The

toolbar buttons have `handlers` that add a new child `Container` into the main container. Last we setup three static `container` instances in the `items` array property **(5)**. Notice that we are not specifying any XTypes for the child items. Unless otherwise specified, all direct children of a Container use "container" as the default XType. When you run the sample, you should get a screen like the left side of figure 4.6 below. Make sure to hit the buttons in the toolbar a few times to see how items are sized once they are added, just like the right side of figure 4.6.

Figure 4.6 The result of a simple default layout. The left side shows the initial state of the code from listing 4.3 while the right side shows the result of clicking the “Add Child” and “Add Fixed Width Child” buttons a few times. The blue border makes it easies to see the bounding boxes for each item.

A change from Sencha Touch 1.x

For those users upgrading from Sencha Touch 1.x, you might be wondering what happened to the `AutoContainerLayout` and `AutoComponentLayout`. The answer is simple; both of them have been replaced by the `default` layout. They have however been added as alternate class names for the default layout to provide backwards compatibility, ensuring you don’t have to change your existing apps to accommodate this change.

Although the `DefaultLayout` provides little to manage the size of child items, it is not completely useless. Relative to the other layouts, it is very lightweight, making it ideal if you

want to display child items that have fixed dimensions. There are times however when you'll want to have child items dynamically resize to fit the parent `Container`. For those days, the `FitLayout` is exactly what you need.

4.2.2 Make it Fit – The FitLayout

The `FitLayout`, shown in the listing 4.4 forces a `Container`'s single child to "fit" to its body element and is, by far the simplest of the layouts to use, only requiring that you have a single item within the `Container`.

Listing 4.4 Fit Layout

```
var myContainer = new Ext.Container({
     fullscreen: true,
     layout: 'fit',
     items:[
        {
             docked : 'top',
             xtype  : 'toolbar',
             title  : 'Fit Layout'
        },{
             xtype : 'container',
             style : 'background-color: pink; padding: 20px;',
             html  : 'I Fit in my parent'
        }
     ]
});
```

The above code generates a simple panel that automatically occupies the entire screen of the device it is run on, just like Figure 4.7.

Figure 4.7 Using the FitLayout to illustrate a single child resizing to fit it's parent.

One caveat to keep in mind about the `FitLayout` is that it is only suitable if a single item is present. The layout will break if multiple items are housed within the same

Container, thus causing undesirable effects. For managing multiple items within a container, take a look at the CardLayout and Box layouts below.

4.2.3 Card Layout

A direct subclass of the FitLayout, the CardLayout combines the concept of components conforming to the size of the Container from the Fit Layout, with the ability to have multiple children under one Container. The layout acts similar to a carousel or wizard, only showing a single "card" or "screen" at a time. Each card automatically occupies the full size of the Container and can house any number of Component instances. Managing which card is shown is done during render time using a containers activeItem configuration, and after rendering, flipping between cards is left to the developer via the containers setActiveItem method. Under the hood, the card layout taps into the containers activeitemchange event to catch whenever it is supposed to change items, and actually perform the change.

Navigation frequently occurs in a wizard-like interface through the use of a previous and next button in a toolbar, or actions embedded within the cards. Of course, there are various other methods to achieve the same; a programmatic timer, or event based being some of them. In listing 4.5 below, we explore how to create a simple wizard style interface.

Listing 4.5 Card Layout

```
var handleNavigation = function(btn) {                                    1

   var currentContainer = myContainer.getActiveItem();                    2

   var indexOfCurrentContainer =
        myContainer. getInnerItems().indexOf(currentContainer);

   var newIndex;

   if (btn.text == "Back") {                                              3
      var newIndex = indexOfCurrentContainer> 0 ?
                     indexOfCurrentContainer-1 :
                     myContainer.getInnerItems().length-1;
   }
   else {
        var newIndex = indexOfCurrentContainer <
                     myContainer.getInnerItems().length-1 ?
                     indexOfCurrentContainer+1 : 0;
   }

   myContainer.setActiveItem(newIndex);
}

var myContainer = new Ext.Container({
   fullscreen: true,
   layout: {
      type: 'card',                                                       4
      animation: 'cube'
   },
```

```
   activeItem: 1,                                                    5
   items: [
      {
        xtype: 'toolbar',
        docked: 'top',
        title: 'Card Layout',
        items: [
           {
                text: 'Back',
                ui: 'back',
                handler: handleNavigation                            6
           },
           { xtype: 'spacer' },
           {
                text: 'Forward',
                ui: 'forward',
                handler: handleNavigation
           }
        ]
      },
      { html: 'Card 1' },
      { html: 'Card 2' },
      { html: 'Card 3' }
   ]
});
```

1. Declare a function to handle the navigation
2. Retrieve the currently active card
3. Determine which button was pressed
4. Give the container a card layout
5. Set the active item, so we start with "Card 2"
6. Attach the navigation handler to the buttons

The first thing we need to do is create a method to control the card flipping **(1)** by determining the active item's index and then calculating which index to show next. This is accomplished by first retrieving the `activeItem` using the `getActiveItem` method **(2)** which returns a reference to the currently active container. From there, we need to determine where in the set of cards the `activeItem` is because the `setActiveItem` method that allows us to navigate between difference cards works based on the index of the new card we wish to move to. To obtain the index, we use the standard array function `indexOf`, feeding it a reference to the `activeItem`. This tells us the location of the card within the items array of the `Container`. By checking the text of the button that was pressed we are able to determine which direction to navigate **(3)**, either back or forward. The only thing left then is incrementing or decrementing the index to reflect the direction we're moving. Since we want to wrap around from the first card to the last, and from the last card to the first, we need to perform an additional check to ensure that the new index doesn't go below 0 or above the amount of items we have in the container. If you don't want to wrap around, you could change this same logic to simply skip the navigation when previous or next is pressed from the first or last card respectively.

Now that we have a way to navigate between different cards in a Card Layout, we actually need to create said cards. To do this, we start with a `Container`, giving it a layout of "`card`" **(4)**. This automatically triggers certain changes in the `Container` to make it listen to the `activeItem` config option, and expose the `getActiveItem` and `setActiveItem` methods. We will utilize the `activeItem` config **(5)** to start with the second card, by setting the config option to 1. This works because cards are in a 0-based index, so index number 1 is actually the second item. After that, we add a toolbar with two buttons, and point the handlers of the buttons to our handy navigation function **(6)** we created earlier. The only thing left to get the results from Figure 4.8 below, is adding the cards as child items. For this we only need to specify the `html` property, since all child items are Containers by default.

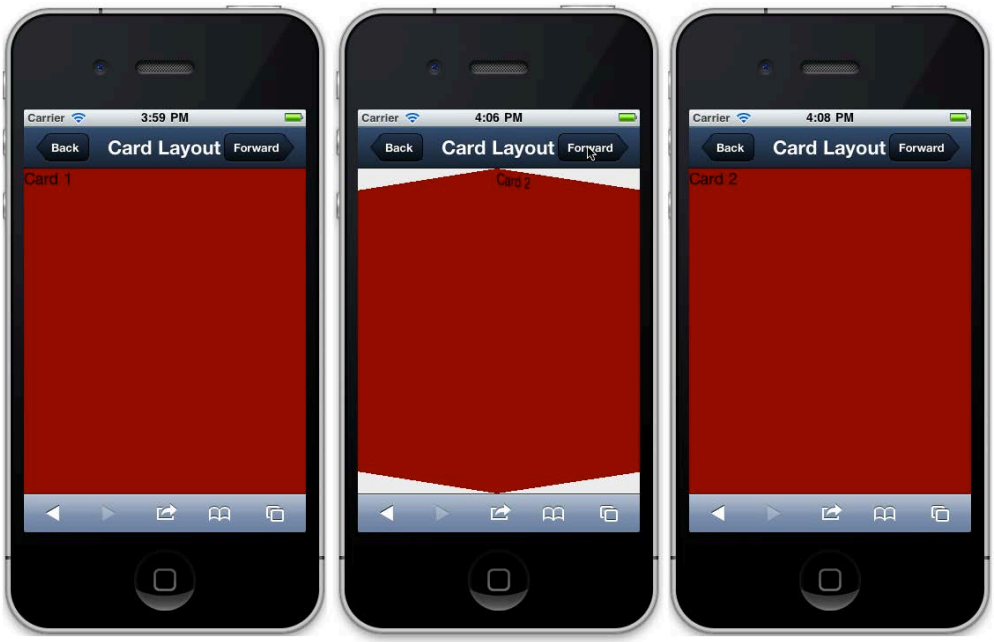

Figure 4.8 A card layout showing the initial card, along with a “cube” transition.

The card layout represents one of the more useful layouts, since it provides a way to show content without much clutter, allowing the user to navigate through it one piece at a time. I do want to express a word of caution here however. Each card takes up valuable rendering time and memory. This is especially true on mobile devices where CPU cycles and RAM are not in abundance. Adding too many cards can result in a significant slow-down of your application. The exact number when this occurs depends on the content of each card,

and is something you will have to play around with. It is certainly something to keep in mind when designing your application.

One of the major drawbacks behind the card layout is that each card automatically occupies the entire container, thus eating up valuable screen-space. To alleviate that, we have another layout at our disposable that allows us to place multiple components on a screen in a row or column like fashion, the Box Layouts.

4.2.4 HBox and Vbox Layout

The HBox & VBox Layouts are different than the one's we've discussed so far, in that they display items in columns or rows respectively. This allows for much greater flexibility when creating complex layouts for your application, as each row or column could utilize it's own layout, allowing you to combine a set of rows with a `CardLayout` in each row for example. Let's dive into the `HBoxLayout`.

Listing 4.6 HBoxLayout

```
var myContainer = new Ext.Container({
     fullscreen: true,
     layout: {
            type: 'hbox',
            pack: "start",
            align: "start"
     },
     defaults: {
            style: "border: 1px solid red;"
     },
     items: [
        {
            html : 'Panel 1', height : 100
        }, {
            html : 'Panel 2', height : 75, width : 100
        }, {
            html : 'Panel 3', height : 200
        }
     ]
});
```

In listing 4.6, you create a `Container` with three irregularly shaped child `Containers`, to properly exercise the different layout configuration parameters available. The first is `pack`, which means "horizontal alignment", and the second is `align` which stands for "vertical alignment". You will notice that the sample code has a `defaults` property that sets a red border style, which will be automatically applied to all child items. This is done so you can better see the boundaries for each item, since `Containers` don't have any borders by default.

The `pack` parameter accepts four different values: 'start', 'center', 'end', and 'justify'. The first three options align the content on the horizontal axis either on the left, the middle or the right side of the `Container`. The fourth option, 'justify', aligns the content with the left

and right side of the `Container` at the same time, thus spreading content across the entire width. Modifying the `pack` parameter in Listing 4.6 will result in one of the rendered panels in Figure 4.9.

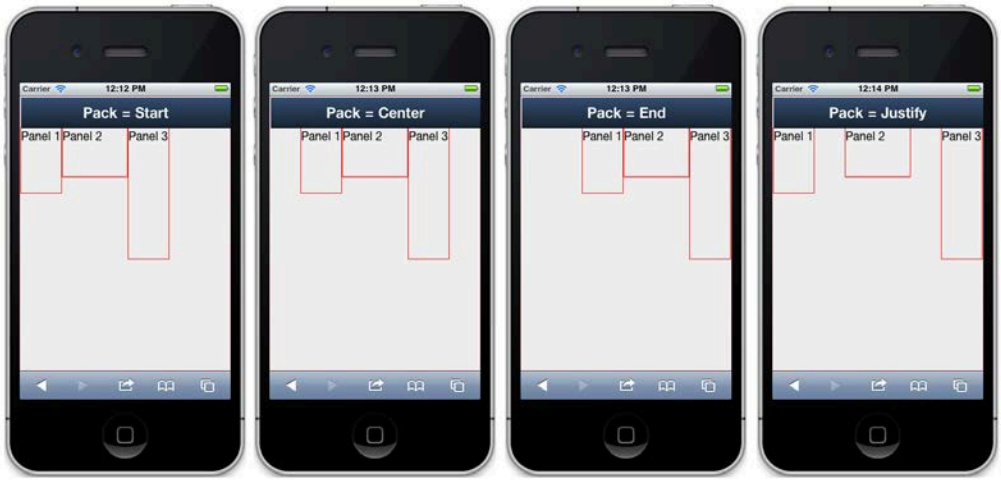

Figure 4.9 Showcasing the different "pack" options. These screenshots use an "align:start" setting as well.

The `align` parameter accepts four possible values: 'start', 'center', 'end', and 'stretch'. The first three options actually align at the top, middle and bottom respectively, and the fourth option overwrites the height, thus stretches the content to occupy the entire height of the `Container`. The default value for `align` is 'center'. Figure 4.10 illustrates how you can change the way children are sized and arranged based on a few different combinations.

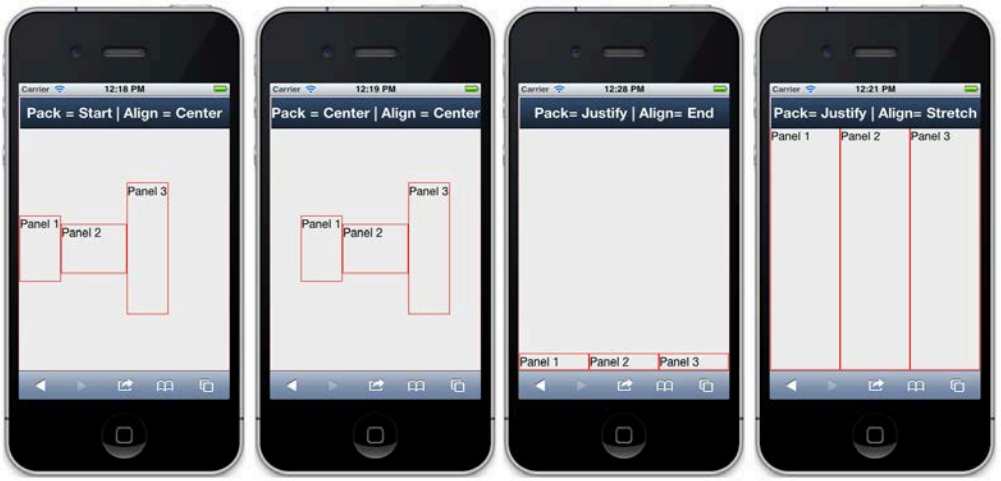

Figure 4.10 Mixing the pack parameter with different align options can produce interesting and useful combinations.

One of the great features behind the box layouts is the ability to size rows/columns either manually, by specifying a fixed height/width like we did in listing 4.6 above, or dynamically via the use of a percentage value or a special `flex` value. The `flex` value acts as a weight, or a priority if you will, instead of a percentage that a `Component` should occupy. Let's assume for a second that you want to create a layout of multiple columns where all columns have an equal width. You would use an `HBoxLayout` and simply give each column the same `flex` value. If you wanted to have two of the columns occupy the first half of the screen, and the third column to expand to the other half, you would have to ensure that the `flex` value for each of the first two columns is exactly half that of the third column. For instance:

```
items: [
     {
            html : 'Panel 1',
            flex :1
     },
     {
            html : 'Panel 2',
            flex :1
     },
     {
            html : 'Panel 3',
            flex :2
     }
]
```

The main caveat when using either one of the box layouts is that each item has to have a sizing indicator. This means fixed size, percentage, or the flex parameter must be present. Furthermore, using the `align:stretch` option only works with `flex`. Leaving these off will either have undesirable results, or simply not work, which can take hours to debug. Trust me, I've been there.

4.2.5 Nesting Layouts

Something many starting developers don't realize is that you can easily nest different layouts to achieve crazy combinations that don't generally exist out-of-the box. Since Sencha Touch doesn't provide for a table layout or column layout like ExtJS does, `HBox` and `VBox` are the only options at your disposal to achieve a multi-row / multi-column layout. In figure 4.11 below, we explore what you can achieve with a little creativity, by nesting the `HBoxLayout` & `VBoxLayout` and adding a modified version from the `CardLayout` sample in listing 4.5 a few pages ago.

Our goal is to create a grid-like structure with a carousel at the top that spans the full width, and two rows of content with three columns each, just like figure 4.11.

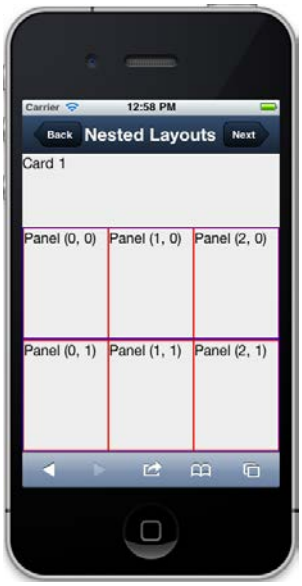

Figure 4.11 It is possible to nest different layouts to achieve more complex results. This could easily turn into a magazine layout, where the cards represent a picture gallery, and the individual panels represent an article.

To get started, we take a standard `Container` instance and give it a `VBoxLayout`. To achieve evenly spaced rows, we specify a `flex` of 1 in the `defaults` property, thus forcing the same height on each row. Notice the additional `width` parameter that forces each direct child to have a width of 100%. This is needed since each child of the main panel does not automatically adjust the width of the child items, and the `flex` parameter only applies to the height since we are using a VBox Layout. Assembled, the code would look like this:

```
   var mainContainer = new Ext.Container({
       fullscreen: true,
        layout: 'vbox',
        defaults: {
               width: '100%',
               flex: 1
        },
        items:[
               // … content items go here …
        ]
});
```

Now that we have our base container, we start by adding content to it, just like listing 4.7 below. First up is the card layout code that we bring over from listing 4.5 and add into the `items` array. The only tweak we have to make to it is to give it an `itemId` **(1)** so we can obtain a handle on it later when we tweak the `handleNavigation` function. After that, we add two panels **(2)** with an `HBoxLayout` and three child panels each. We give each of the panels a one-pixel border to make it easier to see where they start and end. The three child panels within each HBox Container we give a `flex` of 1 and a `height` of 100%, as well as some simple HTML content. The `flex` is needed so the three panels are the same width, and `height` so the panels occupy the full height available. This is exactly the same paradigm as before with the VBox layout, just that the `height` and `width` are reversed.

Listing 4.7 Nested Layouts

```
   var myContainer = new Ext.Container({
        fullscreen: true,
        layout: 'vbox',
        defaults: {
           width: '100%',
           flex: 1
        },
        items:[
           {
                itemId: "cardContainer",                                    1
                layout: {
                   type: "card",
                   animation: 'slide',
                },
                activeItem: 0,
                items: [
                   {
                        xtype: 'toolbar',
```

```
                docked: 'top',
                title: 'Nested Layouts',
                items: [
                   {
                        text: 'Back',
                        ui: 'back',
                        handler: handleNavigation
                   },
                   {xtype: 'spacer'},
                   {
                        text: 'Next',
                        ui: 'forward',
                        handler: handleNavigation
                   }
                ]
             },
             { html: 'Card 1' },                                          3
             { html: 'Card 2' },
             { html: 'Card 3' }
          ]
       },
       {
          layout: 'hbox',                                               2
          style: 'border: 1px solid blue;',
          defaults: {
             style: "border: 1px solid red;",
             flex: 1
          },
          items: [
             { html: 'Panel (0, 0)' },
             { html: 'Panel (1, 0)' },
             { html: 'Panel (2, 0)' }
          ]
       },
       {
          layout: 'hbox',                                               2
          style: 'border: 1px solid blue;',
          defaults: {
             style: "border: 1px solid red;",
             flex: 1,
          },
          items: [
             { html: 'Panel (0, 1)' },                                  3
             { html: 'Panel (1, 1)' },
             { html: 'Panel (2, 1)' }
          ]
       }
    ]
});
```

1. **Specify an itemId to obtain a handle**
2. **Create panels with HBox Layout**
3. **Provide simple HTML content**

To finish it all up, we still need to tweak our handy navigation function for the carousel toolbar to fit into our new sample. The major difference in this version of the function to the one we had before in listing 4.5 is that the card layout is no longer the main container, but instead is nested within another container. This means that we need to obtain a reference to it when we want to interact with it. We accomplish this utilizing the `down` method in conjunction with the newly added `itemId`.

```
var handleNavigation = function(btn) {
     var cardContainer = myContainer.down("#cardContainer");
     var currentPanel= cardContainer.getActiveItem();

     var newIndex;
     var indexOfCurrentPanel =
            cardContainer.getInnerItems().indexOf(currentPanel);

     if (btn.text == "Back") {
            var newIndex = indexOfCurrentPanel > 0 ?
                   indexOfCurrentPanel -1 :
                   cardContainer. getInnerItems().length-1;
     }
     else {
            var newIndex = indexOfCurrentPanel <
                   cardContainer. getInnerItems().length-1 ?
                   indexOfCurrentPanel +1 : 0;
     }
     cardContainer.setActiveItem(newIndex);
}
```

In the end, we have a complex layout that could easily be used as a springboard for an application. Potential uses could include an online magazine or a picture gallery. The possibilities are almost endless.

Now that we have mastered containers and their layouts to some degree or another, it's time we take a look at how Sencha builds on containers, and expands their functionality through two of containers subclasses, the `Panel` and `TabPanel` respectively.

4.3 Floating away… with Panels

Back in the olden days, the Panel class used to be the workhorse widget of Sencha Touch. It was the de facto go-to component whenever you needed to manage multiple child items. These days however, Panels dominance has been usurped by the standard `container`, relegating the `panel` class to a simple extension of `container` with some added styling, like a border, and the ability to float. It is important to note that since Panel does extend Container, it carries with it all of the same abilities that Container does, including the use of layouts, docking items, and managing children. The following is a quick overview of the full Panel inheritance chain.

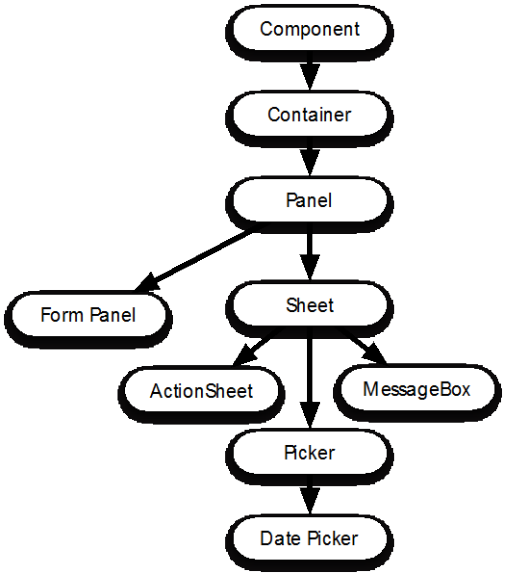

Figure 4.12 The Panel Class Inheritance Model

As illustrated in Figure 4.12, `Panel` is the base for everything floating and modal, such as Sheets, Pickers, and Message Boxes. This is because Panels make for great overlays, as they have special handling for the `showBy` method, whereby the panel shows a small tip pointing to the reference component the panel is shown by. To see an illustration of this, take a peek at Figure 4.13 below.

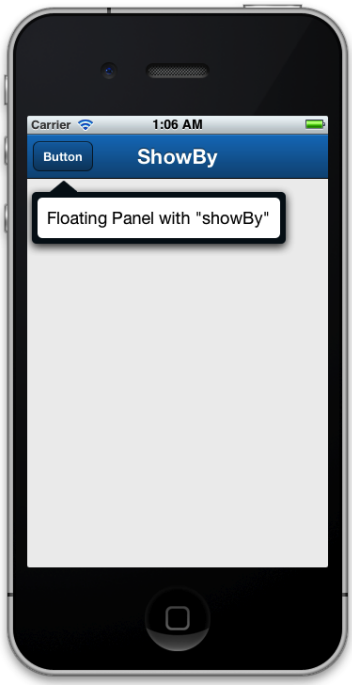

Figure 4.13 Floating panel with 'showBy' used

Although Figure 4.13 does look mighty pretty, you're not here to simply look at pretty pictures. So, let's get our hands dirty and actually implement a `panel`. Since we're touting Panels ability to float, that's what we will cover first. Let's up the ante however, and add the ability to drag panels around as well while we're at it.

4.8 Creating a draggable and floating Panel

```
var floatingPanel = new Ext.Panel({                                    1
    height      : 200,
    width       : 200,
    draggable   : true,                                                2
    floating    : true,                                                3
    html        : 'Some help could go here.',
    left        : 50,                                                  4
    top         : 50,
    items : [
        {
            xtype  : 'toolbar',
            docked : 'top',
            title  : 'Drag me!'
        }
    ]
});
```

```
Ext.Viewport.add(floatingPanel);                                    5

var fsPanel = new Ext.Panel({
    fullscreen : true,
    style      : 'background-color: #CCF;',
    html       : 'Full screen Panel'
});
```

1. Create the floating Panel
2. Make it draggable
3. Make it floating
4. Move to the 100 X and Y coordinates.
5. Show the panel by adding it to the viewport.

As exhibited in Listing 4.8, to create a floating and draggable `Panel` **(1)**, we must set the `draggable` **(2)** and `floating` **(3)** options to Boolean `true`. Setting `floating` to true instructs `Panel` to set its element as absolutely positioned on the page immediately after it has rendered on screen. Likewise upon render, the `Panel` enables drag and drop by means of creating a new instance of `Ext.util.Draggable` for itself.

KNOW THE ROOT OF THIS FUNCTIONALITY

Even though we are using Panel to demonstrate floating and the drag and drop features, the root of this functionality is the `Ext.Component` class. This means that you can enable these features for just about any subclass of `Component`, which includes any custom widgets that you create. All of the features we'll be discussion moving forward will be from the perspective of Panel.

Next, we set its position **(4)** to 50 pixels in the X and Y coordinate space. This demonstrates that the absolute positioning set forth by the `floating` configuration option works.
After creating and rendering the floating and draggable `Panel`, we need to add it to the Ext.Viewport instance **(5)** that is automatically created, in order to make the panel show up. This is because the after the Panel is created, it is not managed by any parent container, and since it is floating, we need to use the main viewport to make it visible.

This listing concludes with the creation of a full screen `Panel` for contrasting purposes. We did this because the probability that you'll have a floating and dragging `Panel` above a full screen and stationary `Panel` is very high.

Below illustrates how it renders on a phone.

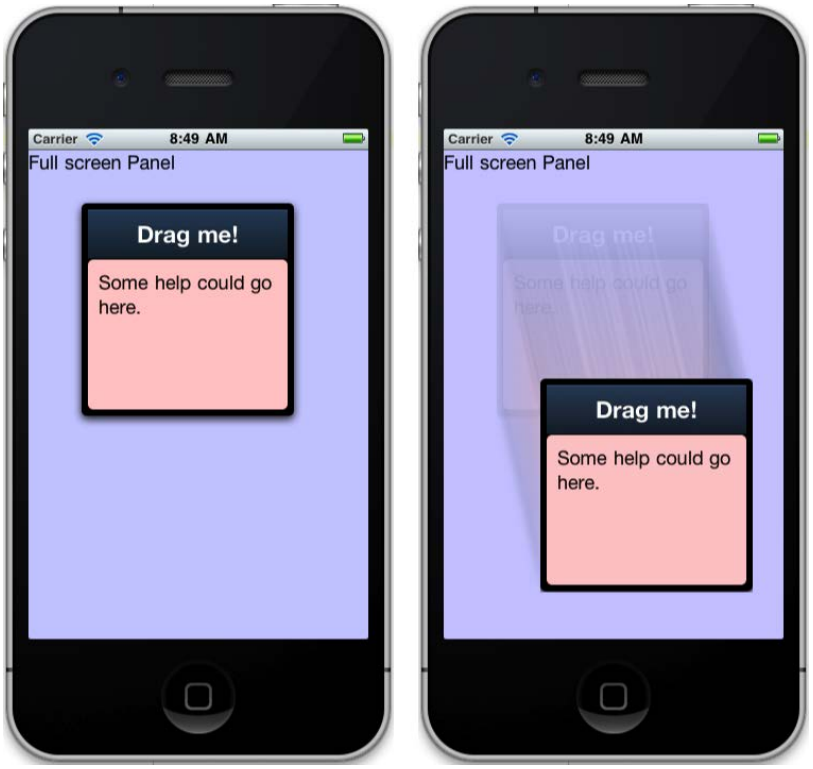

Figure 4.14 Demonstrating the floating and draggable Panel.

After rendering our floating and draggable Panel on Screen, you might notice that the floating Panel looks more decorated than a non-floating Panel. This is because the Panel class wraps an extra thick border around itself when it's set to floating. It gives it more of a Windowed UI feel.

Because we set `draggable` to true, you can move the `Panel` around the screen by means of a drag gesture. Notice how the Panel is constrained to the limits of the display. This is a typical touch interface UI convention.

Post-instantiation floating and dragging.

If you want to enable these features after a Component has been instantiated, you can use the `setFloating` and `setDragging` methods. The framework API has full details on how these methods work under the `Ext.Component` class.

Before we wrap up this discussion, there are a few important nuggets of information you should know about floating Panels.

The first is the fact that panels that are floated can be set to modal, meaning that Sencha Touch will create div element underneath the floating Panel effectively creating a dimming effect on the rest of the screen. This helps drive attention to your floating Panel. To enable the modal feature, add the following configuration parameter to Listing 4.8.

```
modal: true
```

Here is what a modal floating Panel looks like compared to a non-modal floating panel.

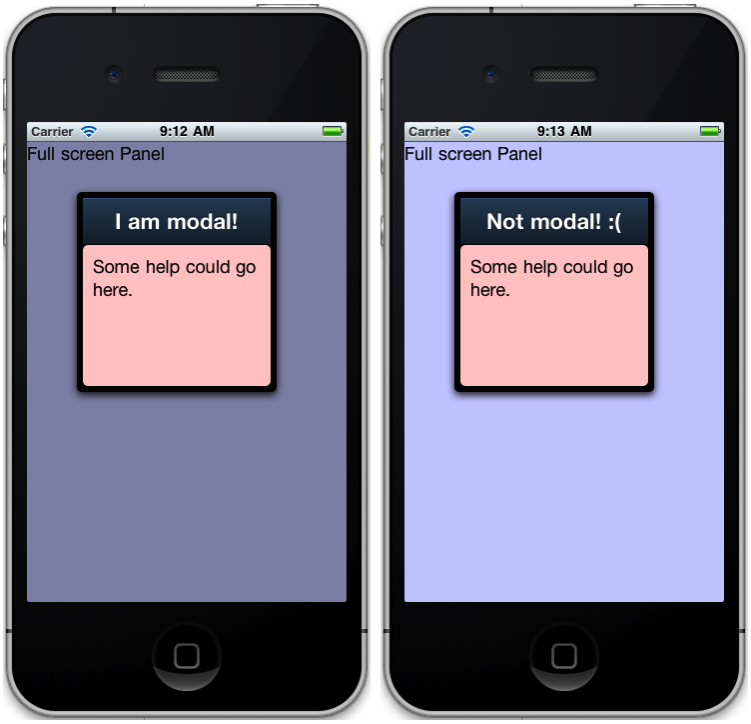

Figure 4.15 Comparing a modal floating Panel (left) to a non-modal floating Panel (right).

Figure 4.15 illustrates how a modal floating `Panel` (left) has a dimming effect on the rest of the screen, making everything else but the `floating` Panel look darker compared to the non-modal floating `Panel`, which seems to have the same contrast as the rest of the UI.

The next tidbit you should know about floating `Panels` is that it will self-hide whenever the user taps outside of the confines of the `Panel`. This is the default behavior of a modal Panel, but can be modified via the following parameter.

```
hideOnMaskTap : false
```

The `hideOnMaskTap` configuration property defaults to Boolean `true`, which enables the default behavior. To alter the default, set it to `false`. Doing so will ensure that the Panel will not hide automatically when tapping outside of the `Panel`'s element. One thing to note is that this default behavior is enabled whether or not you set `modal` to `true` or not.

DON'T LEAVE YOUR USERS HANGING!

I've run into situations where developers introduced bugs, where they disabled `hideMaskOnTap` and gave the users no way to dismiss the dialog. Remember, if you're going to disable `hideOnMaskTap`, your users will have no way to get rid of the modal Panel. This means you'll need to create a button that allows them to do so, or hide the modal Panel after a certain time.

The feature that you should know is that the appearance and disappearance of the Panel can be animated relatively easily. All you have to do is set the following attributes.

```
showAnimation : 'pop',
hideAnimation : 'fade'
```

The `showAnimation` configuration property is used by the Panel to animate the appearance of the Panel, while `hideAnimation` is used to animate the disappearance. Each of these parameters accepts a String, Object. You would pass a Boolean false if you wanted to disable a `Component`'s default transition. For instance, if you created a new instance of `Ext.MessageBox` and didn't want to use its animation for show and hide actions.

Use a String where you want to use any one of the pre-configured animations from the `Ext.anims` Singleton. The best ones to use are `pop`, and `fade` because they are designed to transition in a single item, relative to the `cube` or `flip` animations, which are designed to animate between two different components. If you only set a `showAnimation`, Sencha Touch will automatically do the opposite animation for hiding the `Panel`.

Animation performance is crucial.

Be aware that animations use CSS3 transitions and work best on devices that have a dedicated GPU and are complimented with an instance of mobile webkit compiled to utilize the GPU. Enabling animations on devices that cannot handle them will result in a clunky UI, and ultimately turn off your users.

We've covered a fair bit about the first of our two container extensions, the Sencha Touch Panel class. We learned about floating, and dragging, and showing a panel by another component. As promised, let's delve into the second of the extensions, the Tab Panel.

4.4 Flip the deck with TabPanels

The TabPanel is a special widget that although the name might imply differently, extends Container and is a widget that allows you to easily create an interface for users to navigate between screens by tapping on a special Button known as a Tab located in a special Toolbar called the TabBar. The TabPanel mimics the traditional desktop tabbed interface but within the modern mobile UI guidelines.

To fully understand how they work and how they can be customized, we'll have to create a base example that we can modify later on.

Listing 4.9 Creating a simple TabPanel.

```
new Ext.TabPanel({                                                    1
    fullscreen          : true,
    ui                  : 'light',
    tabBarPosition      : 'top'                                       2
    items : [                                                         3
         {
            style  : 'background-color: #FCC;',
            html   : 'Panel 1',
            title  : 'Users'                                          4
        },
        {
            style  : 'background-color: #CFC;',
            html   : 'Panel 2',
            title  : 'Admins'
        },
        {
            style : 'background-color: #CCF;',
            html  : 'Panel 3',
            title : 'Locations'
        }
    ]

});
```

1. **Create the TabPanel**
2. **Configure the TabPanel**
3. **Add the TabPanel child items**
4. **Set the title for the Tab**

Listing 4.9 represents an example of a simple TabPanel **(1)** that contains four child Panels **(3)**. We utilize the tabBarPosition **(2)** to set whether the tab bar should show at the top or the bottom of the tab panel. In this case, we specifically defined it as "top", but given that this is the default, we could have just left it off as well. Lastly, we configure an array of items **(4)** for the TabPanel to manage. These are generic configuration items that will be used to create instances of Container. What is important to focus on is the title **(5)** property. This is not a normal property for Container. In this case, it will end up being used to set the title text of instances of Tab that will live in the TabBar.

Here is what all of this looks like rendered on screen.

Figure 4.16 The results of Listing 4.9, a rendered TabPanel.

Figure 4.16 illustrates the result from Listing 4.9, which contains our simple `TabPanel` with left aligned `Tabs`. To display any of the screens, the user can simply tap on any of the `Tabs`, which will use the default 'slide' animation transition.

If we wanted to change the alignment of the tab bar items to be right aligned for example, we would have to utilize the `tabBar` config option to override the default alignment. To do this, we would simply add the following to Listing 4.9

```
tabBar: {
     layout: {
            pack: "right"
     }
}
```

The `layout` parameter might look immediately familiar from earlier parts of the chapter. If so, you're absolutely correct, since the tab bar uses an HBox layout under the hood. So, by adding `pack: "right"` we simply override a portion of the layout to align items differently.

The same paradigm would be true if we wanted to change the animation used when switching between cards. Let's say we wanted a cube animation, instead of the default slide. In that case, we would add the following to Listing 4.9.

```
layout: {
     animation: "cube"
},
```

The `animation` configuration property is part of the layout options for the tab panel, and instructs the `TabPanel` to animate the transition between the different cards (screens). We set this property to use the 'cube' animation, which is different from the default 'slide' animation. To disable animations, you set this property to `false`. Generally, you want to consider disabling `animation` when animations become sluggish.

See all of the animations.

You are given the choice of 'fade', 'slide', 'flip', 'cube', 'pop' and 'wipe' animations to use for `TabPanel` transitions. You can see all of these animations in the "Kitchen Sink" example, which can be found in the examples directory of the Sencha Touch SDK.

You might notice that when top-aligning the `TabBar`, the rendered `Tabs` only show the given text, and look rather plain. Given the way `Tabs` are constructed from a class perspective, you cannot easily add icons to them without an override, extension or plugin of some sort. This means you will either have to go through quite a bit of pain to hack those in your self, or simply bottom-dock the `TabBar`, which gives you the ability to set an `iconCls` property for each child item in the Tab Panel. To better understand this, I've included an example that modifies Listing 4.9 to do just that.

Listing 4.10 Showing a bottom-docked TabBar for a TabPanel.

```
new Ext.TabPanel({
    fullscreen          : true,
```

```
    ui                  : 'light',
    tabBar              : {
        docked : 'bottom',                                          1
        layout : {
            pack : 'center'
        }
    },
    items : [
         {
            style   : 'background-color: #FCC;',
            html    : 'Panel 1',
            title   : 'User',
            iconCls : 'user'                                        2
        },
        {
            style   : 'background-color: #CFC;',
            html    : 'Panel 2',
            title   : 'Groups',
            iconCls : 'team'
        },
        {
            style   : 'background-color: #CCF;',
            html    : 'Panel 3',
            title   : 'Locations',
            iconCls : 'maps'
        },
        {
            style   : 'background-color: #FFC;',
            html    : 'Panel 4',
            title   : 'Settings',
            iconCls : 'settings'
        }
    ]
});
```

1. **Docking the TabBar to the bottom**
2. **Setting the iconCls**

To dock the `TabBar` to the bottom of a `TabPanel`, we provide an alternative to the `tabBarPosition` we used before, by injecting a `docked: bottom` configuration **(1)** into the `tabBar` config, thus piggyback on the fact that we are already using the `tabBar` config. Next, we add an `iconCls` property **(2)** for the items in the tab panel to dress them up a bit.

Figure 4.17 illustrates the newly configured `Tabs` rendered in the bottom-docked `TabBar`.

Figure 4.17 Our TabPanel with a bottom-docked TabBar containing Tabs that display icons.

As illustrated in Figure 4.17, the bottom-docked `TabBar` renders with the `Tabs` displaying the icons we configured for them. There is one point that I need to make before we wrap this conversation up.

It's important to note that the `iconCls` configuration property is required for `Tabs` bottom-docked `TabBars`. If you don't specify the `iconCls`, the `Tabs` will render, but be barely usable as they will be extremely small, just like Figure 4.18.

Figure 4.18 Bottom tabs without icons

There you have it! We've just seen how we can configure TabPanels with both top and bottom-docked TabBars. You should be able to use these lessons when developing your TabPanel-enabled application.

4.5 Summary

We've covered a lot of ground in this chapter; from the way Containers manage child items, and how to add and remove components dynamically in an already rendered UI. You also learned how to arrange Components in different ways using the various layout schemes. While we covered enough material to make just about any developer's head spin, we barely scratched the surface of what is possible.

That being said, you may have noticed a recurring theme across many of the samples we encountered in this chapter. More specifically, we used toolbars and docked items in quite a few of them. Given the prevalence of toolbars and docked items in even our simple examples, it is important for us to better understand the capabilities of these powerful tools at our disposal. So, with the next chapter, we will do exactly that, by delving deeper into Toolbars, Buttons, and Docked Items.

5

Toolbars, Buttons, and Docked Items

This chapter covers

- Unlocking the secrets of Docking
- Mastering the Toolbar
- Using and customizing Buttons

In the last chapters, we covered a lot of foundational topics, including the `Component` lifecycle, `Containers` and Layouts. Many of the examples encountered thus far made use of docked items via toolbars, perfectly setting the stage for you to master the topics in this chapter such as, toolbars, buttons, or anything else you want to dock.

Since we're already aware of the fact that Containers and Panels have the ability to dock widgets, we will focus our efforts on expanding on that concept, by learning how to dock just about any widget, to any of a container's outer quadrants. In the process, we will discover the importance placed on the order of docked items. We will follow that up with a variety of flavors to organize `Buttons` and other widgets inside of the `Toolbar`. There you'll learn how to use the Spacer component as well as customize the `Toolbar`'s `HBoxLayout` implementation.

We'll end the chapter by unraveling the secrets of `Buttons`, and how to customize their appearance by exploiting the built-in configuration options. Although this chapter is a bit shorter and doesn't cover quite as many topics as the others, the experience gained will be absolutely necessary for almost any application you build.

5.1 Looking into docked items

While it's relatively easy to create a Container with items docked to a quadrant, there are some hidden nuances that I've discovered along the way. To illustrate these we will need to create a relatively basic example to play around with.

5.1.1 Understanding the basics

As with most things in life, a picture is worth a thousand words. To get a better understanding of the basic example we're going to build, take a quick peek at the illustration below, which represents of our very simple example of a docked `Container` on each outer quadrant of a `Panel`.

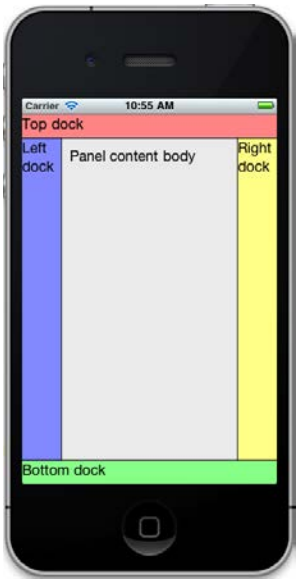

Figure 5.1 A simple Panel example with four docked Containers.

Figure 5.1 illustrates docked containers at each possible outer quadrant of a `Panel`. A `Panel` (or any `Container` really) will allow any amount of docked items at any given time. For learning purposes, we are only going to four, one in each quadrant.

Here is what the code looks like to achieve the result in Figure 5.1.

Listing 5.1 Placing containers at each of the outer Panel quadrants.

```
var topDock = {                                                    // 1
        xtype  : 'container',
        docked : 'top',                                            // 2
        style  : 'border-bottom: 1px solid; background-color: #F99;',
        height : 100,
```

```
            html   : 'Top dock'
        },
        bottomDock = {
            xtype  : 'container',
            docked : 'bottom',
            style  : 'border-top: 1px solid; background-color: #9F9;',
            height : 100,
            html   : 'Bottom dock'
        },
        leftDock = {
            xtype  : 'container',
            docked : 'left',
            width  : 100,
            style  : 'border-right: 1px solid; background-color: #99F;',
            html   : 'Left dock'
        },
        rightDock = {
            xtype  : 'container',
            docked : 'right',
            width  : 100,
            style  : 'border-left: 1px solid; background-color: #FF9;',
            html   : 'Right dock'
        };

    var myPanel = Ext.create("Ext.Panel", {
        fullscreen  : true,
        bodyStyle   : 'padding: 10px;',
        html        : 'Panel content body',
        items : [                                                    // 3
            topDock,
            bottomDock,
            leftDock,
            rightDock
        ]
    });
```

{1} Configure docked item objects
{2} Top-positioned docked item
{3} Register the docked items with Panel

Listing 5.1 contains the necessary code to dock a container at each of the `Panel`'s outer quadrants. It begins with registering references to configuration objects **(1)** for each of the docked `Containers`.

> **Just about anything can be docked.**
>
> We're using the Container in Listing 5.1 to reduce code complexity of this example, but you can pretty much place any widget to a `Panel`'s outer quadrant.

In order to control where a widget is docked, we must set a `docked` property **(2)**. There are four possible values, `top`, `bottom`, `left` and `right`. Each value controls which outer quadrant to render the widget in.

Lastly, we register the docked `Containers` with the newly created Panel by setting the `Panel`'s `items` property to an array **(3)**, containing references to each of the configuration objects we created earlier. Notice that even though the docked items are within the items array, they show up in their docked position, and don't interfere with the panel content body.

Docking Multiple Items in a Quadrant

Although our examples only dock a single item into each quadrant, you could easily duplicate one of the configuration objects in the `items` array, and thus dock multiple items in one of the quadrants. The additional items would simply stack. Keep in mind however that the order matters.

We should now have a good hold on how to register docked items. One thing that might be a bit confusing is that the order in which you register `docked` items will adversely affect how the items are laid out in the quadrants. Take the following configurations for instance.

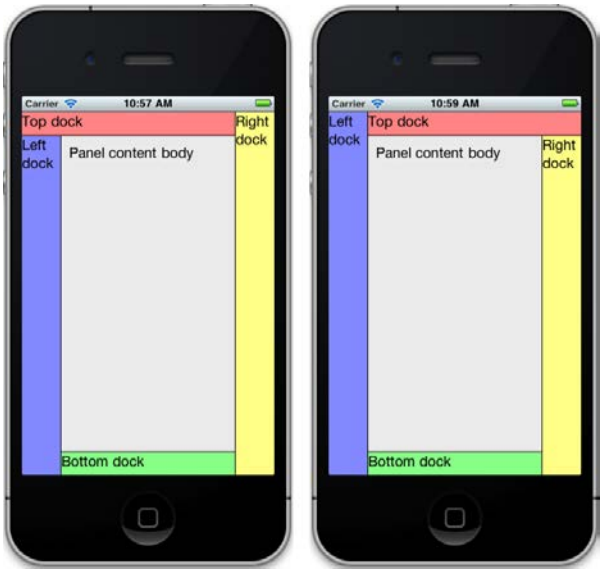

Figure 5.2 The order in which you place docked items will affect the amount of screen space they have

Figure 5.2 demonstrates dock configurations that are different compared to the example we created in Listing 5.1. This is because the order in which we list docked items controls how they are sized. The rule for sizing is simple: the docked items with the lower indexes get the largest portion of a quadrant.

To test this rule, we'll have to modify the `items` Panel configuration property in Listing 5.1 and shuffle the order of the docked items as follows:

```
items : [
    leftDock,
    rightDock,
    topDock,
    bottomDock
]
```

With the left and right docked items listed first, the docked containers would render as such.

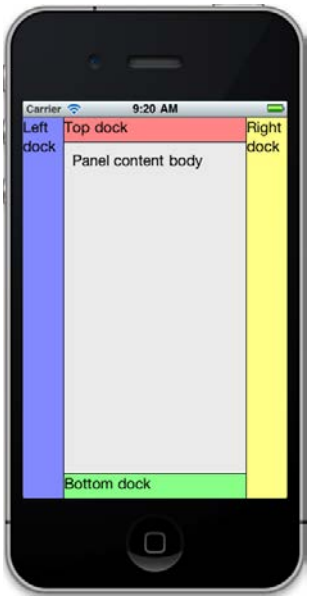

Figure 5.3 The result of our first modification to Listing 5.1.

As predicted, the left and right docked containers are larger than the ones located at the top and bottom. What if we wanted the left and right items to be the two that are the largest? We would again have to modify the order in which we list `items`. Given what we've just learned, can you predict how we should organize the `items` to achieve this desired result?

If you guessed the items in the order listed below, you are absolutely correct.

```
items : [
    leftDock,
    topDock,
    rightDock,
```

```
        bottomDock
    ]
```

The rendered result of our second change to Listing 5.1 results in the docked items rendering like the following illustration.

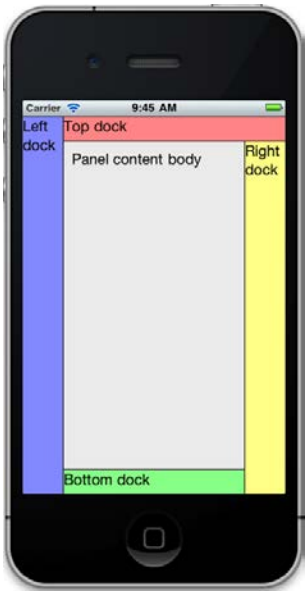

Figure 5.4 The result of our second modification to Listing 5.1.

As predicted, the left dock is the largest of the four and the top takes as much space as it can, while the right captures all available space in its quadrant. The bottom dock is the absolute smallest.

One secret to how this all works is that the order of precedence rule only applies to docked items that are intersecting.

For instance, the top intersects with the left and right docked, but never the bottom. The bottom dock could care less about how large the top is. Likewise, the right dock intersects with the top and the bottom docks. The right doesn't have any introspection to the left dock, nor does it care that the left dock is even rendered on screen.

Use our docked items testing tool

To test out every possible configuration combination, you can use our dockedItems configuration tool at

http://senchatouchinaction.com/examples/chapter04/dock_configurator.html

We have a solid grasp on how the docked items order of precedence rule works, but we're not quite done with them yet. To fully realize their power, we should look at how we can leverage the dynamic nature of docked items to create dynamic UIs that can expand the workflows of your application.

5.1.2 Dynamic docking

In Chapter 4, we learned how to dynamically add and remove items in a `Container` via the `add` and `remove` methods. Adding docked items works exactly the same. In fact, it uses the same methods (add and remove) as any other components would. The only difference is that docked items must have the docked property set. The add and remove methods will automatically figure out if the item is a docked item, and do its thing.

> **For those developers upgrading from Sencha Touch 1.x**
>
> In previous iterations of the framework, you had to employ the `addDocked` and `removeDocked` methods to make changes to docked items. With Sencha Touch 2.x, these methods have been deprecated, and their functionality has been rolled into the standard `add` / `remove` methods.

Containers (and thus Panels as well) expose two convenience functions to get a hold of a specific docked item, or all of them at once.

Table 5.1 The methods used to manage docked items dynamically.

Method	Description
`getDockedComponent`	This method is responsible for retrieving and returning a reference to a Component that is in the docked collection. It takes a single argument, the `itemId` (String) of the component or the index (integer).
`getDockedItems`	This returns a list of all currently docked items.

These two methods are particularly useful if you need to work with docked items, as they allow you to surgically target a very specific docked item, or work with all of the docked items at once, without having to worry that non-docked items will interfere.

Listing 5.2 demonstrates adding and removing top docked items by means of a general-purpose button rendered inside of a `Panel`.

Listing 5.2 Adding and removing dockedItems.

```
var myPanel = new Ext.Panel({                                    // 1
    fullscreen  : true,
    bodyStyle   : 'padding: 10px;',
```

```
    layout      : { type  : 'vbox', pack  : 'center' },
    items       : [
        {
            xtype   : 'button',
            text    : 'Add top',
            handler : handleAddButton
        },{
            xtype   : 'button',
            text    : 'Remove top',
            handler : handleRemoveButton
        }
    ]
});

function handleAddButton() {                                          // 2
    var dockedItems = myPanel.getDockedItems();

    myPanel.add(                                                      // 3
        {
            xtype  : 'container',
            docked   : 'top',
            style  : 'border-bottom: 1px solid; background-color: #F99;',
            height : 30,
            html   : 'Top dock: ' + dockedItems.length
        }
    );
};

function handleRemoveButton() {                                       // 2
    var dockedItems = myPanel.getDockedItems();

    if (dockedItems.length > 0) {
        var dockedItem = myPanel.getDockedComponent(dockedItems.length-1);
        myPanel.remove(dockedItem, true);                             // 4
    }
};
```

1. **Create a Panel with add and remove buttons**
2. **Function to handle button clicks**
3. **Add a top docked item**
4. **Remove the last top docked item**

Listing 5.2 contains the code to render a general purpose Panel with two buttons in the middle. One button to add a new top docked item, and one button to remove the last top docked item. Here's how this all works.

We begin by creating a `Panel` **(1)** with two `buttons` defined as simple child items within the panel. Notice that we are using a `vbox` layout with `pack: center`, which will vertically align the buttons in the center. The first button is for adding top docked items; the second button is for removing docked items. Each of the buttons has a `handler` defined which points to a function to handle the tap event. The `handler` for the add button **(2)** starts out by getting a hold of all the docked items. We need to do this, since our example allows for adding multiple docked items, and we want to be user friendly and allow the user

to tell the different items apart. We do this, by dynamically tweaking the configuration for the items we are adding **(3)**, and setting the `html` of the new item based on the count of docked items. The configuration object specifies `docked: top`, which causes the newly added item to be stacked in the top quadrant. To actually add the top docked item, we simply use the `add` method of the panel.

The remove handler in turn works by first getting all of the docked items. We then check to see if any exist, and if so, retrieve the last docked item. Since the docked items use an array, which is a 0-based index, the `length` of the array will always be one more than the index of the last item, hence the need to use length-1. We use the `getDockedComponent` method in conjunction with the index to retrieve the last docked item, which is then removed by passing the item to the `remove` method **(4)** of the panel. You may notice the additional Boolean we pass to the remove method. This is to ensure that the docked item is actually destroyed, and not just removed from the Panel. In case this doesn't ring a bell, a revisit to chapter 4 might be necessary.

Here is what it looks liked rendered on screen.

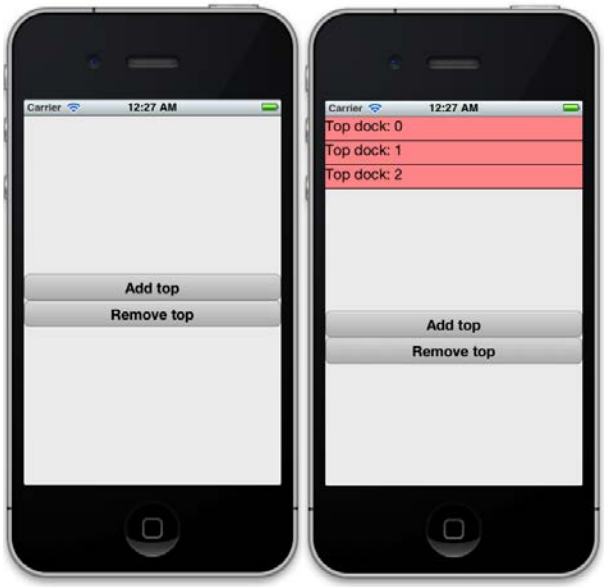

Figure 5.5 The initial results of Listing 5.2 (left) and adding the top docked Container by means of tapping the button (right) a few times.

As illustrated in Figure 5.5, we can see that a Panel renders with two buttons in the center of the screen (left). A couple taps of the button adds and docks a `Container` to the top dock, demonstrating the `add` method in conjunction with the `docked` config. Any subsequent

taps on the remove button will remove one of the top-docked containers, which demonstrates the `remove` function.

> **A much more elaborate example**
>
> Much like our dock configuration testing tool, we have an example that adds and removes docked items to all quadrants. To see this example, please visit http://senchatouchinaction.com/examples/chapter04/add_remove_docks.html

We've successfully demonstrated how to add, remove as well as query for docked items. Next up, we will elaborate on one of the most commonly docked items, the toolbar.

5.2 Gearing up the Toolbars

The Toolbar is a widget that extends `Container`, and is typically docked at the top or bottom, and is used to house a set of buttons and text elements, such as a centered title. It is however more versatile than that, and can in fact be docked anywhere, and contain pretty much anything; not just buttons and text.

5.2.1 Under the hood

The secret to the `Toolbar`'s versatility is that it is a subclass of `Container`, which means that it can render and manage just about any `Component` in the Sencha Touch framework. Before we get ahead of ourselves however, let's take a step back, and begin with a basic example to build upon.

Listing 5.3 Constructing a simple Toolbar

```
var toolbar = {                                                    // 1
    xtype  : 'toolbar',
    docked : 'top',
    title  : 'User admin'                                          // 2
};

new Ext.Panel({
    fullscreen  : true,
    style       : 'background-color: #CCF;',
    html        : 'Full screen Panel',
    items       : [
        toolbar                                                    // 3
    ]
});
```

#1 The Toolbar configuration object
#2 The Toolbar title
#3 Include the top-docked Toolbar

Listing 5.3 contains a recipe for a `Panel` that includes a top-docked `Toolbar` **(1)**. To keep things simple for now, we've only given the toolbar a `title` **(2)**, and nothing more. Since

we're going to be building upon this Listing, we're going to use a Tablet to view it, as it will give us the most amount of screen real estate to work with.

Here is how our freshly minted toolbar looks when rendered on a Tablet.

Figure 5.6 A simple Toolbar with a title rendered in a tablet.

Figure 5.6 shows us how something as simple as adding a Toolbar with a title can easily dress up an application. While there are many situations where having a `Toolbar` with just a title is sufficient, more often than not, you will run into the situation where you need buttons for users to interact with as well. This is where things get interesting as Toolbars can contain a mixture of Buttons, text and much more.

5.2.2 Adding buttons to a Toolbar

Listing 5.4 contains a modification of Listing 5.3, where we redefine the Toolbar configuration object by adding two `Buttons` and a `Spacer` component.

Listing 5.4 Adding buttons to our simple toolbar.

```
var toolbar = {
    xtype  : 'toolbar',
    docked : 'top',
    title  : 'User admin',
    items  : [
        {
            xtype : 'button',                                    // 1
            text  : 'Submit'
        },
```

```
        {                                                          // 2
            xtype : 'spacer'
        },
        {
            xtype : 'button',
            text  : 'Cancel'
        }
    ]
};

new Ext.Panel({
    fullscreen  : true,
    style       : 'background-color: #CCF;',
    html        : 'Full screen Panel',
    items : [
        toolbar
    ]
});
```

#1 Configure a simple button
#2 Add a Spacer Component.

Listing 5.4 contains a Toolbar with a title, two `Buttons` **{1}** and a `Spacer` component **{2}**. To drill down into what all of this is doing under the covers, we must first look at how it's rendered on screen.

Figure 5.7 Rendering a toolbar with a title, two Buttons and a hidden spacer Component.

Figure 5.7 illustrates how the newly modified top-docked toolbar renders on screen. Notice how the Submit and Cancel buttons are on opposite ends of the screen. This is because of

the Spacer component taking 100% of the available screen space between the two Buttons.

To get a full grasp what's going on here, I'll need you to recall your experience from Chapter 4, where you learned about the HBoxLayout and the flex configuration property. By default, the Toolbar utilizes the HBoxLayout internally, whenever it is docked at the top or bottom. The reason the buttons are pushed to opposite ends is because the Spacer component is between them, and it has a default flex property of 1, instructing the HBoxLayout to size the Spacer to 100% of the available horizontal space minus the widths of the buttons and some padding.

5.2.3 Centering items

The fact that the Toolbar implements an HBoxLayout can be extremely useful if you wanted to center buttons or a set of them.

Listing 5.5 modifies Listing 5.4 and is a recipe for centering an instance of SegmentedButton between our two previous Buttons.

Listing 5.5 Centering with two Spacer components

```
var toolbar = {
    xtype  : 'toolbar',
    docked : 'top',
    items  : [
        {
            xtype : 'button',
            text  : 'Submit'
        },
        {                                                    // 1
            xtype : 'spacer'
        },
        {
            xtype : 'segmentedbutton',                       // 2
            items : [
                {
                    text : 'Comedy'
                },
                {
                    text : 'Thrillers'
                },
                {
                    text : 'Horror'
                }
            ]
        },
        {
            xtype : 'spacer'                                 // 3
        },
        {
            xtype : 'button',
            text  : 'Cancel'
        }
    ]
```

```
};

new Ext.Panel({
    fullscreen  : true,
    style       : 'background-color: #CCF;',
    html        : 'Full screen Panel',
    items       : [
        toolbar
    ]
});
```

#1 The first Spacer
#2 Configure the SegementedButton
#3 The second Spacer

Listing 5.5 demonstrates how to center a SegementedButton **{2}** by sandwiching it between two `Spacer` **{1}{3}** components. Here is how it looks rendered on the screen.

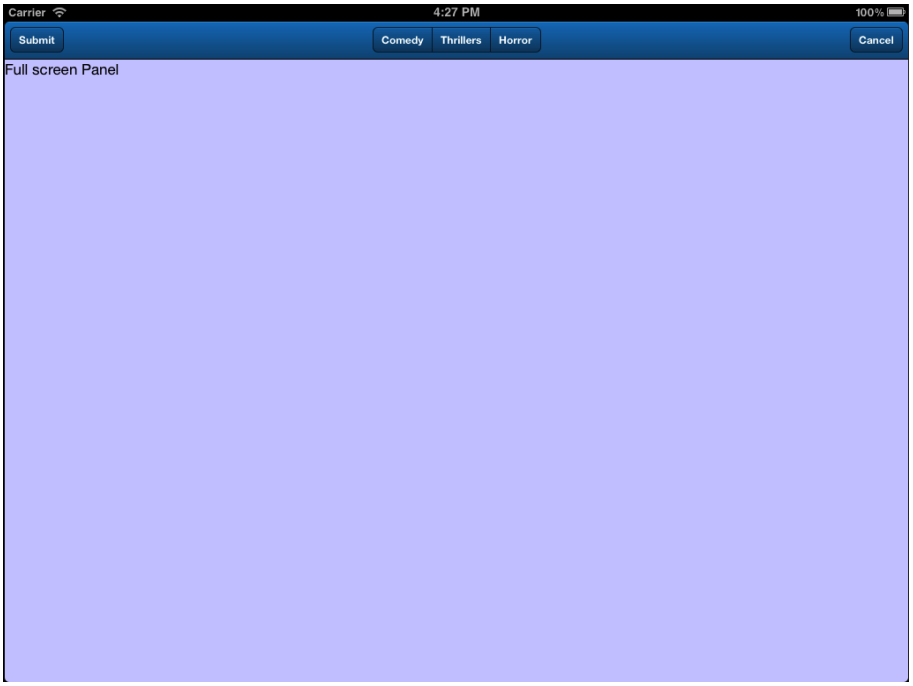

Figure 5.8 Centering a Segmented button by means of two Spacer components.

The results of Listing 5.5 is show in Figure 5.8, where a `SegmentedButton` is centered between two outside `buttons`. The reason it's centered is because the Spacer has a default `flex` of `1`, instructing the `HBoxLayout` to allow the spacers to equally utilize all of the available space.

Because of the HBoxLayout's flexibility, you can set static widths on the Spacers or even have the buttons dynamically sized via the flex parameter. The HBoxLayout's flexibility can lead to truly creative Toolbar item organization, but as I said earlier, you can render just about anything inside of the Toolbar.

What we just implemented was the centering of items in addition to having items flush on each side of the Toolbar. If you wanted to have all items centered, you wouldn't need to use the flexible Spacer component, but instead simply instruct the HBoxLayout to do the work for you by adding the following parameter to your Toolbar configuration object.

```
layout: {
    pack : 'center'
}
```

We've just experimented with various ways to layout items inside of Toolbar. Next, we'll take a stab at a Toolbar configuration that can implements more than just Buttons and Spacers.

5.2.4 Adding non-standard Components

Since we've already learned that the Toolbar extends Container and thus allows for any item to be placed in it, we're going to proceed with a recipe for a Toolbar-based search for your application. To do this, we'll need to inject a Text input field into the Toolbar items configuration parameter.

Listing 5.6 Adding a Text input field to a Toolbar.

```
var toolbar = {
    xtype  : 'toolbar',
    docked : 'top',
    title  : 'User admin',
    items  : [
        {
            xtype : 'spacer'                                    // 1
        },
        {                                                       // 2
            xtype : 'textfield',
            width : 200
        },
        {
            xtype    : 'button',                                // 3
            iconCls  : 'search',
            ui       : 'plain',
            iconMask : true
        }
    ]
};

new Ext.Panel({
    fullscreen  : true,
```

```
        style       : 'background-color: #CCF;',
        html        : 'Full screen Panel',
        items       : [
            toolbar
        ]
    });
```

#1 Flexible Spacer
#2 A TextField config object
#3 The search button

Listing 5.6 validates that you can render more than just `Buttons` in a `Toolbar`. Because a common design paradigm is to place search fields to the right of the `Toolbar`, we utilize a Spacer component to move things over. Next is the `Text` **{2}** input field, followed by a `Button` **{3}** configuration option that has some unique styling.

The below Figure illustrates how it renders on screen.

Figure 5.9 The Text input field rendered inside of a Toolbar alongside a nicely styled button.

So far, we've explored a lot of how the `Toolbars` work, and we have not talked much about `Buttons`. It's hard to argue with the fact that `Buttons` are an integral part for many of the screens we'll be creating for any application.

This is why we'll be shifting gears to focus on these versatile widgets.

5.3 Go ahead. Press my Button!

The Sencha Touch `Button` class is used all over the place in our applications, and for whatever reason, its versatility is often overlooked by many of us. The fact of the matter is that the `Button` can be customized really easily to match almost all of your application needs.

The general use of `Buttons` is pretty simple. For example, here is a typical `Button` configuration object.

```
var myBtn = {
    xtype     : 'button',
    text      : 'Delete',
    iconCls   : 'delete',
    iconMask  : true,
    ui        : 'drastic',
    scope     : someObject,
    handler   : function() { /* code here */ }
}
```

The above configuration object can be used to render a Button within a `Toolbar` or `Container`, though the `xtype` property is typically omitted when being used inside a `Toolbar` because its default type is 'button'.

When tapped, the `Button` will execute a function, known as a handler, which can be executed within the scope of any object.

> **Does "scope" sound foreign?**
>
> In JavaScript, functions can be executed within the "scope" or context of any Object. This means that when the function is executed, the magic keyword "this" points to said Object. I have a great slide deck demystifying the "this" keyword, which can be found at http://www.slideshare.net/moduscreate/javascript-classes-and-scoping.

At run time, you can change the handler of any button via its `setHandler` method, which takes two arguments; the first of which is required is a reference to function that will be called upon the `Button`'s tap event, as well as the `scope` in which that function will execute.

5.3.1 Customizing Buttons

Buttons can be rendered anywhere on screen, and can be children of any Container or subclass. Here is an example of some modified buttons rendered in a top-docked Toolbar, as well as inside of a Panel. Due to the sheer amount of configuration properties to set this example up, it is going to be pretty long, so please bear with me.

Listing 5.7 Customizing buttons.

```
var tbar = {                                                    // 1
    xtype    : 'toolbar',
    docked   : 'top',
    defaults : {
        iconMask  : true
    },
    items : [                                                   // 2
        {
            text      : 'delete',
            iconCls   : 'delete',
            iconAlign : 'left',
            ui        : 'back'
        },
        {
            text      : 'organize',
            iconCls   : 'organize',
            iconAlign : 'right',
            ui        : 'decline'
        }
    ]
};

var bbar = {                                                    // 3
    xtype    : 'toolbar',
    docked   : 'bottom',
    height   : 60,
    defaults : {
        iconMask : true
    },
    items : [                                                   // 4
        {
            iconCls   : 'refresh',
            iconAlign : 'top',
            ui        : 'plain'
        },
        {
            text      : 'search',
            iconCls   : 'search',
            ui        : 'drastic',
            iconAlign : 'bottom'
        }
    ]
};

new Ext.Panel({
    fullscreen  : true,
    defaultType : 'button',
    defaults    : {
    iconMask : true
    },
    layout : {
        type : 'vbox',
        pack : 'center',
        align: 'center'
```

```
    },
    items : [
        tbar,                                                    // 5
        bbar,
        {                                                        // 6
            text      : 'generic',
            width     : 150
        },{
            text      : 'compose',
            iconCls   : 'compose',
            iconAlign : 'right',
            ui        : 'action',
            width     : 150
        },{
            text     : 'star',
            iconCls  : 'star',
            ui       : 'confirm',
            width    : 150
        }
    ]
});
```

#1 The top toolbar
#2 Buttons for the top-docked toolbar
#3 Bottom-docked Toolbar
#4 Add buttons to the bottom Toolbar
#5 Including Top and bottom Toolbars
#6 Adding Buttons to the Panel body

Listing 5.7 contains the necessary code to create a `Panel` that contains both bottom and top-docked `Toolbars` as well as three `Buttons` rendered in its body. Here is how it all fits together.

We begin the listing by creating a configuration object for the top-docked `Toolbar` **{1}**. This toolbar contains two buttons **{2}** with an icon alignment set via the `iconAlign` property.

> **Know thy iconAlign.**
>
> The `iconAlign` property can be set to 'left', 'right', 'top' or 'bottom'.

Because we are using the built-in icons, both of the Buttons **must** have the `iconMask` option set to `true`. Setting this to `true` instructs Sencha Touch to apply the default `iconMaskCls` CSS class ('x-icon-mask') to the Button, applying a CSS alpha channel mask that makes the icon visible on the screen. If not set to true, the icons will not be visible.

> **Learn about CSS alpha channel masks.**
>
> CSS alpha channel masks, allow you to take one image (the alpha channel), and blend with another to create complex blending that previously required powerful image editing

software. The following page is an excellent beginners tutorial on masks. http://www.webkit.org/blog/181/css-masks/ .

Next, we configure the bottom-docked `Toolbar` **{3}** that also has two `Buttons` **{4}**. One of the buttons has its icon top-aligned, while the other is bottom-aligned. As we'll see in a little bit, the icons will render above and below the text, according to the `iconAlign` configuration property. Notice that the `iconMask` property is set to `true` via the `Toolbar`'s `defaults` configuration object.

UNLEASHING THE PICTOS (ICONS).

A little known fact is that Sencha Touch has over 330 icons that can be used in your mobile apps, but only 26 of them are actually usable out of the box. The access the rest, you'll need SASS and the related libraries, such as Ruby and some elbow grease. To download SASS, you can visit http://sass-lang.com/. Also, Shea Frederick, a veteran of the Sencha community, has an excellent article on his blog that details how to use SASS to compile the Sencha Touch CSS to include only the icons you plan on using in a process he calls "trimming the fat". His article can be found at http://www.vinylfox.com/custom-styling-a-sencha-touch-app/.

The last block of code in this listing is a full screen `Panel` **{5}**, that contains three `Buttons` **{6}** centered on screen by means of the `VBoxLayout`.

There are some other bits of info about this listing that we have to discuss, but I think it's best to see the result first, as that will better frame up what we'll be talking about next. Here is how it all looks rendered on a mobile device.

Figure 5.10 Rendering customized buttons as a result of Listing 5.7.

Figure 5.10 illustrates the result of Listing 5.7, rendering a Panel containing a top-docked and a bottom-docked Toolbar with two customized buttons each, as well. Notice how the icon alignment is organized exactly how we configured it to.

Each of the buttons has something unique aside from their `iconAlign`-ment. And that has to do with the `ui` configuration property. Of the five buttons rendered on screen, one of them is a generic button, meaning it does not have an icon and does not have a `ui` configuration property to alter its appearance.

This optional configuration property is important to remember because it allows you to alter appearance of the Buttons and other widgets with ease. Here is how it works.

When you specify the `ui` configuration property, Sencha Touch adds a CSS class to the widgets element, allowing you to alter its physical appearance by using some of the built-in 'ui' styles. The two buttons in the top-docked Toolbar in Figure 5.10 have `ui` properties set. The first of which is 'back', alters the physical shape of the button but is the default color. The second button uses the 'decline' `ui` style giving it the red color, but does not have its shape altered.

If you look at the rendered Buttons in the Panel, you'll see that there are three colors. The first is a generic Button and demonstrates how Buttons will appear if rendered outside of a toolbar without icons or a ui style applied. The second uses a ui style called 'action', giving it a blue tint, while the third button uses the 'confirm' ui style, coloring it green.

Focusing on the bottom toolbar, you'll find a Button that has no border around it and a generic Button with the iconAlign set to 'bottom'. The focal point here is the first button, which has its ui property set to 'plain'. Setting a Button's ui property to 'plain' within a toolbar will render the button without borders. I must stress the fact that the 'plain' ui style only works within Toolbars. Adding this ui style to a button rendered inside of a container will prevent the buttons from rendering.

In addition to the 'back' ui style, you can alter the shape of the of the buttons via the 'forward', 'round', and 'small' ui styles. You even have the capability to use compound styles, such as 'action-small', 'confirm-small' and 'decline-small'.

Instead of listing all of the possible combinations, I have prepared a quick example on all of the possible out of the box styles, which is illustrated below.

Figure 5.11 Demonstrating all possible Button styles.

Figure 5.11 shows all of the possible out of the box Button ui styles within a Toolbar and a Panel. The ui style used to alter each Button is set as its text, so you can quickly figure out what style you want to use. This example is live and can be accessed via this url: http://senchatouchinaction.com/examples/chapter04/all_button_styles.html.

You now know just about everything you need to know about how buttons work and how to deploy them in a normal or customized manner.

5.4 Summary

Great work! We covered a lot in this chapter and set a solid foundation for using many of the building blocks for mobile application development with Sencha Touch.

We began by looking into some of the intricacies behind docked items, and where they can be placed, as well as the importance that the order of docked items has on their sizing. From there, we continued on with the most commonly docked item, the `Toolbar`, and learned how to arrange and space out items within toolbars. Last but not least, we explored how to customize buttons, and make them a bit more user friendly through the various icons and styling options. All of this work set the stage for us to explore how Sencha Touch uses many of these concepts in the build-in widgets it ships with. Our next chapter focuses on `Sheets`, `Pickers`, and the `MessageBox`, all of which utilize various concepts from this, and the past chapters.

6

Getting the User's Attention

This chapter covers

- Creating simple overlays
- Utilizing Sheets as a modal dialog for user interaction
- How to use the Picker class
- Using and customizing Message Boxes

While exploring the exciting world of containers, panels and multi-screen applications via the use of carousels and tabs in the previous chapter, you might have noticed that all of those paradigms share one commonality: forcing the user away from the current application screen to facilitate user interaction. Although this is frequently a desired behavior, especially if content needs to be shown in a logically separated way, navigating between different screens can disrupt the flow and usability of your app. This is particularly true if all you really need is to get a quick message to the user, or prompt for simple information like a name. In such cases, using a `TabPanel` or a `CardLayout` that navigates to a different screen, simply to return to the user after information has been gathered is just overkill.

This is exactly where `Sheets`, `Pickers`, and the `MessageBoxes` come into play. These components are specifically designed to allow quick interaction with the user and allow you to easily overlay information onto the current screen (`Sheet`), allow the users to select something from a list of values (`Picker`), or notify the user of something that has happened (`MessageBox`). Each component serves its purpose, and understanding the in's and out's of all three components and how they fit together is advisable, and is exactly the reason why we will start our journey through these components by taking a look at `Sheets` first.

6.1 Using sheets for modal user interactions

Sheets form the basis for all modal dialogs in Sencha Touch. Whether you're looking at a `DatePicker`, a `SelectField`, a `MessageBox`, or an `ActionSheet`, they all extend the basic `Sheet` class. To see exactly how all the different modal dialogs are based on the `Sheet` class, take a look at the class diagram in figure 6.1.

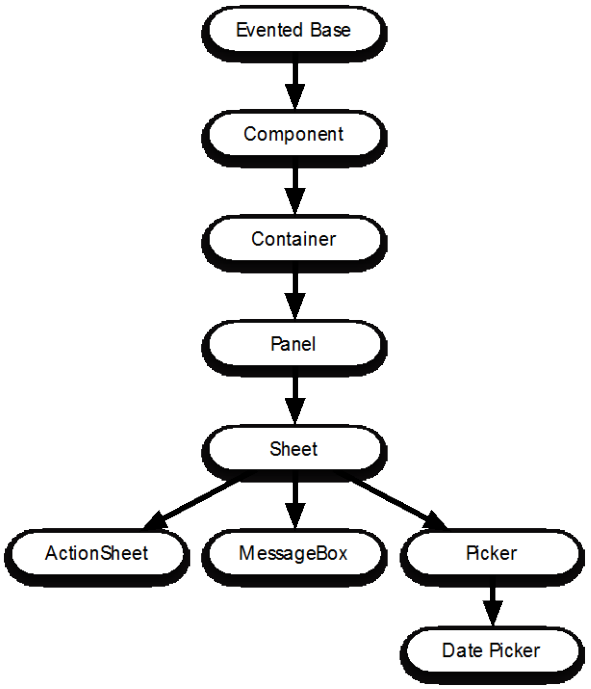

Figure 6.1 – Sheets & Modal Dialogs class diagram

FOR THOSE DEVELOPERS COMING FROM EXTJS

You can think of `Sheets` as a close cousin to the `Ext.Window` class with a modal option set to true. The main difference being that `Sheets` are actually `Panels` and as such bring with them all the options that standard `Panels` bring with them. This includes layouts, toolbars, and managing of child items.

By default, sheets slide in from the bottom of the screen, automatically covering up existing content and imposing a mask on the rest of the screen. Although this is the default, we are not limited to this behavior only. With a few simple tweaks to the configuration, the `Sheet`

can be made to enter from any direction using the `enter` property, thus causing it to slide in from the left side of the screen for example. The beauty behind `Sheets` is that no matter which behavior you chose, the `Sheet` will behave the same way on all devices, causing it to work seamlessly on phones and tablets alike. Take a look at figure 6.2 below to see a sample of a fully rendered `Sheet` instance on a phone and a tablet.

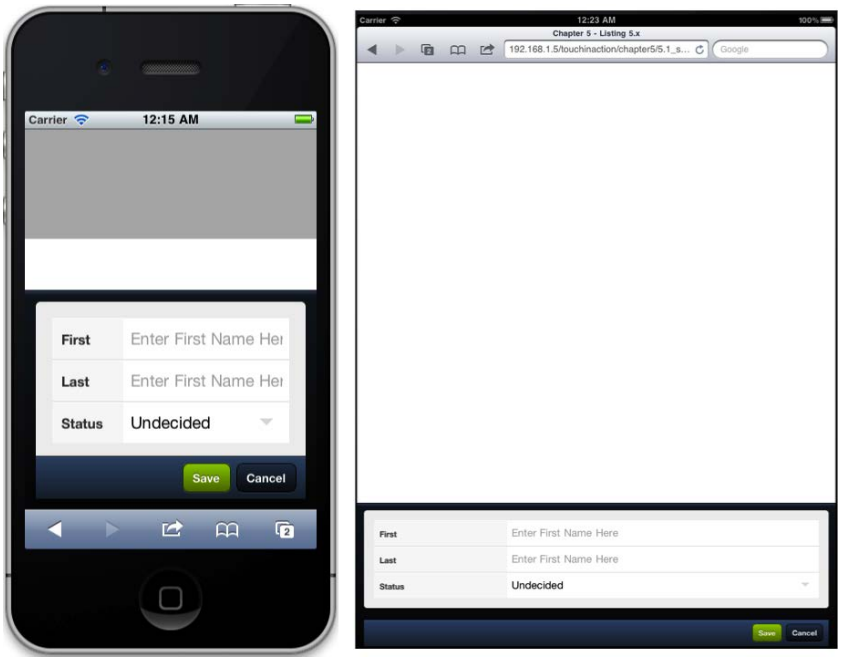

Figure 6.2 – Left side illustrates a sheet sliding in from the bottom on a phone; right side shows the same sheet rendered on a tablet.

As you can see, the exact same sheet rendered on two different devices shows the same content in the same place, only utilizing more space on a tablet than on a phone. Reorienting a device keeps the sheet in its place and automatically recalculates the size of the sheet.

"Size" is where one of the weaknesses of `Sheets` comes to light. Unless you specify a `height` and `width` parameter, or utilize the stretching configuration options (`stretchX` and `stretchY` respectively), your sheets will not automatically size themselves properly. Since sheets are `Panels`, they use the default `AutoContainerLayout` we talked about in Chapter 3, rendering components one on top of the other without any care about sizing or restrictions on the height of the `Sheet`. Leaving off the `height` for example, can result in a

sheet that looks like figure 6.3, where the content is simply not shown, and all that is visible is the toolbar.

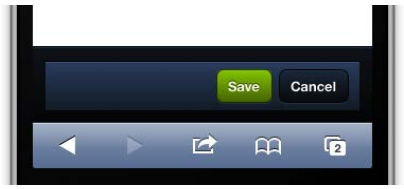

Figure 6.3 A sheet without proper height

On the flip side, adding too many elements without specifying a proper `layout`, can result in one of the two scenarios from figure 6.4 below, where the content either extends beyond the sheet, or part of the sheet reach beyond the screen, inaccessible to the user.

Figure 6.4 Sheets without proper layout configuration.

The moral of the story is that you should always make sure you specify the proper `height`, `width`, and `layout` configuration to avoid situations like this.

This is enough theoretical talk for the moment. It's about time we get our hands dirty and start building some sheets.

6.1.1 Using Sheets for simple overlays

At its most basic function, a Sheet can be used as a simple overlay, triggered to show additional information about content on the screen.

FOR THOSE DEVELOPERS COMING FROM EXTJS

You can think of `Sheets` as a more gimpy version of the powerful `Ext.Tooltip` class that exists on the ExtJS side. This is mainly because on the ExtJS side, things like mouse overs are possible, and 'close' buttons are built in from the start, in an aesthetically pleasing way. Sheets in Sencha Touch in comparison are bound to finger input, and require you to add your own close button and style it.

In general, it would probably be advisable to use a floating panel to represent the overlay, since that provides functionality for "show by", which aligns the floating panel with a specified component or element. There are however certain instances where design might dictate that overlays are shown by sliding in from the bottom of the screen (like sheets do) for space consideration, or any other arbitrary reason. Listing 6.1 illustrates how to go about such a setup.

Listing 6.1 – Using a Sheet as an overlay

```
var myPanel = Ext.create('Ext.Panel', {                                    1
     fullscreen: true,
     html: "Test the overlay by <a href='#'>clicking this link</a>",
});

myPanel.element.on(                                                        2
     "click", showOverlay, this,
     {
            delegate: "a",                                                 3
            preventDefault: true
     }
);

function showOverlay() {
     var newSheet = Ext.create('Ext.Sheet', {                               4
            stretchX: true,
            height: 130,
            hideOnMaskTap: true,
            enter: "top",
            hidden: true,
            items : [
                {
                    html: "Place your additional information right here.<br
/>It can be multi line and everything!"
                }
            ]
```

```
    });

    Ext.Viewport.add(newSheet);                                    5
    newSheet.show();
}
```

1. **Define application panel with content**
2. **Handle click events on the panel body**
3. **Only handle clicks on anchor tag**
4. **Create the sheet**
5. **Add the sheet to the viewport before showing it**

Probably expected more, eh? Fortunately, the code ends up on the simple side, starting out with a full screen `Panel` **(1)** containing some simple HTML content. You probably noticed the anchor tag with an empty `href` in the content. This anchor doesn't actually go anywhere; instead we are going to utilize it to trap the user's click, and only show a `Sheet` when the anchor was clicked. To do this, we setup an event handler on the `Panels` body `element` **(2)**. Notice that we have to use the `element` property to get a handle on the panel body. The click handler is relatively standard, defining the event we are trapping, the function to call (`showOverlay` in this case), and the scope of the function. What might be new is the object that comes after the scope. Here we are setting a `delegate` **(3)**, which filters the click event to only fire if an element that matches the delegate is clicked. Delegates in this case follow standard CSS selectors, meaning our particular `delegate` is setup to only listen for an "a" (anchor) tag to be clicked. Thus we conveniently ignore clicks to any other part of the `Panel` body.

Utilizing Delegates to filter your click events

Delegates allow you to filter click events to only respond when a DOM element matching the specified delegate is clicked. In general, delegates follow a format similar to that of simple CSS selectors, providing you with options to either generically trap clicks on a particular HTML tag at large (like an <a> or <p>), or to be more specific and only trap clicks for elements with a specific CSS class by dot notation like "a.myclass" (where myclass represents a CSS class).

Once a click occurs, we need to show the `Sheet` of course, which happens in the `showOverlay` method **(4)**. The `Sheet` is setup to slide in from the top, which is accomplished by setting the `enter` property. Changing this property to "left", "right", or "bottom" changes where the `Sheet` enters the screen. To ensure the sheet occupies the full width of the screen, we set it to stretch on the x-axis via `stretchX: true`. For content we simply populate it with some text through the `html` property. By default, the sheet doesn't understand when it needs to hide. This means you either have to add a toolbar with a close button on it, or set it to automatically hide whenever the mask (the area no occupied by the sheet) is clicked via the `hideOnMaskTap` property. Notice that after the sheet is created, we

actually need to add it to the overall Viewport **(5)**, before it can be shown. This is required so the sheet can be managed for layout purposes. One thing to keep in mind here is that once we add the sheet to the viewport, it would normally show up automatically (without any animation). To fix this, we set the sheet to `hidden: true`, in order to give us full control over when it should be shown.

All in all, the setup is deceptively simple. Where things get interesting however is when dealing with content that needs to be dynamically loaded into a sheet. In that case, the elements in your panel body would each need a unique identifier so you can determine which specific element was clicked. From the event handler, you would make your Ajax call to the backend to retrieve whatever data you need, waiting for the success handler to come back before showing the content of the sheet. This way, no sheet would show if the Ajax call fails, and you can fish the content out of the response and dynamically set it before creating the sheet. Et voila!

Although using `Sheets` as overlays is a nice example, it most likely represents one of those cases you don't run into all too often. One of the more common uses for sheets is the Action Sheet, which mimics the iOS way of prompting users to make a choice.

6.1.2 Using Action Sheets

Whether you need to prompt the user for a "yes/no" type answer, or really really confirm that the user wants to delete that one record in a list, Action Sheets are the preferred way to handle this type of user interaction. From a design perspective, Action Sheets take after the standard Apple iOS style prompts and will look immediately familiar to most iPhone/iPad users. Even if you're developing for non-apple devices, fret not, as the design is simple and intuitive enough to be self explanatory to the user. Evidence of that is figure 6.5 below.

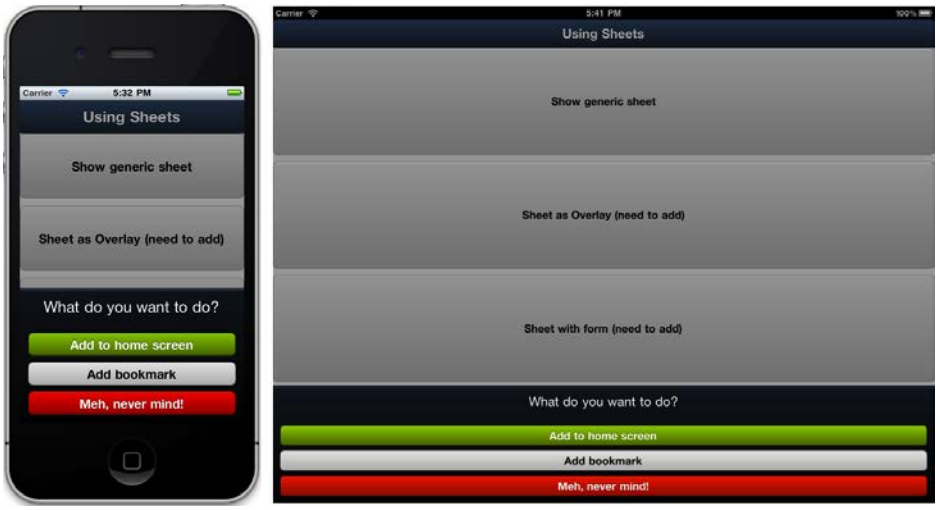

Figure 6.5 – Simple action sheet with three choices and a title.

By default, action sheets always slide in from the bottom, and always occupy the full width of the screen, and just like the normal `Sheet` class, `ActionSheets` do not change their look or behavior based on the type of device the user has. This means that the action sheet from the left side of figure 6.5 above will look exactly the same on a tablet, but simply be really wide, as is evident on the right side of figure 6.5. While this might seem a bit counterintuitive at first, especially given the aesthetics of the tablet version, it does create a sense of consistency across different devices, and removes any doubt as to the meaning of the action sheet.

Setting up an action sheet is relatively straight forward, and listing 6.2 below illustrates how we would create the action sheet from figure 6.5 above.

Listing 6.2 – Simple Action Sheet

```
var myActions = Ext.create('Ext.ActionSheet', {
    items : [
        {
            xtype  : 'component',                                          1
            height : 50,
            style  : 'color: #FFF; text-align: center; font-size: 1.2em;',
            html   : 'What do you want to do?'
        },
        {
            text : 'Add to home screen',                                   2
            ui   : 'confirm'
        },
        {
            text : 'Add bookmark'                                          2
        },
```

```
        {
            text : 'Meh, never mind!',                                    2
            ui   : 'decline'
        }
    ],
    defaults  : {                                                         3
        handler : function() {
            this.ownerCt.hide();
        }
    }
});
Ext.Viewport.add(myActions).show();
```

1. **Use a simple component + html for a title**
2. **Define the actual actions**
3. **Setup a default handler for all action buttons**

We begin by setting up an `Ext.ActionSheet` instance with a title and three items. In this particular implementation, we are opting to put the title in a standard `component` **(1)** using `html`, and placing it in the items array of the Action Sheet, instead of a docked title bar.

The visual difference might not be immediately apparent, until you compare the title bar from figure 6.5 with that of figure 6.4. A docked title bar creates a starker visual separation between the title and the content, primarily due to the different background and spacing between the toolbar and the content. By utilizing a simple `component`, we are sacrificing the functionality that a `Toolbar` would provide us, in lieu of the title looking more like it is part of the content.

Next we continue by adding the three actions **(2)** into the `items` array. Since the default item type for `ActionSheet` is a `Button`, and all actions are buttons, we don't need to specify an `xtype` for our actions. Each action receives a title, via the `text` property, and an optional `ui` parameter to visually distinguish it. Notice that none of the buttons have a `handler` defined. In this state, they would simply be dummy buttons that don't do anything. We take care of that through the use of the `defaults` object, and provide a default `handler` **(3)** for all actions, which simply hides the `ActionSheet`. Since the `handler` is not scoped in any specific way, the handler function will receive the default scope, the button that clicked it. Since we don't want to hide the button, we simple utilize the convenient `parent` property, which gives us access to the button's parent, the `ActionSheet`. Keep in mind that this particular setup is simply for illustrative purposes. In a real world application, each item would most likely have a unique handler that performs a particular action, instead of all actions sharing the same handler.

One of the caveats to keep in mind when dealing with `ActionSheets` is that you can define an arbitrary amount of actions and no limit is enforced. While this is great in cases where you need to present a handful of options, it can quickly get out of hand if your actions exceed the available screen space, as the Action Sheet does not support scrolling by default.

In cases where you need to present a particularly large set of items to the user, simply use a `Picker`, just the thing we will cover in the next section.

6.2 Choosing Pickers

Imagine sitting at the Caesar's Palace in Las Vegas; drink in one hand, and the other feeding a one-armed bandit. You pull down its arm, and the slots start spinning. The first slot shows a seven; the second seven shows. One more and you hit the jackpot. The third slot is spinning for what seems like an eternity. It stops, and ... you lose yet again.

`Pickers` in Sencha Touch follow a very similar concept as gambling style slot machines; they provide a series of "slots" with a predefined set of values for each slot. But unlike Vegas, Sencha Touch allows users to actually pick the exact value they want. For a sample of this, take a look at the following screen.

Figure 6.6 – A typical picker with a single "slot"

The `picker` behaves just like all other classes in the `Sheet` family, where the default behavior is to slide in from bottom of the screen when triggered, and occupy the full width of the screen, behaving the same way on phones and tablets alike. Internally, Sencha Touch uses a `Picker` for the `Select` form field, which we will cover in more detail in chapter 7. In order not to get ahead of ourselves though, we should first explore simple pickers like the one above.

6.2.1 Creating a Simple Picker

Under the hood, the Picker is comprised of a `Sheet` with multiple `Ext.Picker.Slot` instances, where each `Slot` is a scrollable `DataView`. A picker can have an arbitrary amount of slots, only truly limited by the amount of screen real estate available to you. Listing 6.3 demonstrates how to go about building a picker like figure 6.6 above.

Listing 6.3 – A simple Picker

```
var myPicker = Ext.create('Ext.Picker', {
    useTitles : true,                                          1
    hidden    : true,
    slots     : [
        {                                                      2
            name  : 'agegroup',
            title : 'Age group (years old)',
            data  : [                                          3
                {text: '1 - 10', value:  1},
                {text: '11 - 14', value: 2},
                {text: '15 - 18', value: 3},
                {text: '19 - 24', value: 4},
                {text: '25 - 31', value: 5},
                {text: '32 - 40', value: 6},
                {text: '41 - 50', value: 7},
                {text: '51 - 75', value: 8},
                {text: '75 - 100+', value: 9}
            ]
        }
    ]
});
Ext.Viewport.add(myPicker).show();
```

1. **Have the Picker generate titles for us**
2. **Setup a slot in the Picker**
3. **Provide data for the Slot**

We start out with the simplest `Picker` possible, containing only single `Slot`. We instantiate the `Picker`, and tell it to give each column a title **(1)** via the `useTitles` property. When this is enabled, the picker will automatically add a `Toolbar` to each `Slot` with a title equal to the `title` property defined in the slot. In our case, the title would be "Age group (years old)". We continue by providing the one and only slot **(2)** defined for this particular picker. The `Picker` automatically provides the XType for the `Slot`, so we don't have to worry about it at all. Giving the `Slot` a name provides us with a way to identify each slot and its corresponding value if we were to call the `getValue` method of the `Picker`. The only thing left is to define the data to show. This is done via an array of objects **(3)**, each of which consists of a `text` property and a `value` property. The former being the actual text that shows in the picker, the latter being the value that would be returned as part of the `getValue` call.

If we wanted to add another slot to the picker, we would simply build it adjacent to the current one, and structure it the same way, with a name, a title, and an array of data objects. That's all there really is to pickers.

When looking at the documentation for Pickers in the Sencha Touch docs, you will most likely notice a severe deficiency in styling options. The standard picker looks somewhat bland. One low hanging fruit, however, is that you can easily add HTML tags to the text. This allows you some quick and dirty, albeit not dynamic, styling for each item. A sample of that would look like this slot definition:

```
{
     name : 'color',
     title: 'Color',
     data : [
         { text: '<div class="color-red">RED</div>', value: 'red' },
         { text: '<div class="color-green">GREEN</div>', value: 'green' },
         { text: '<div class="color-blue">BLUE</div>', value: 'blue' }
     ]
}
```

This snippet of code assumes that the following CSS classes are present:

```
.color-red { background: red; }
.color-green { background: green; }
.color-blue { background: blue; }
```

In this particular case, each item in the slot would show with a different color, red, green, and blue respectively, just like figure 6.7.

Figure 6.7 – Picker with styled html content

Although you might cry foul, claiming that this is cheating and doesn't really style the picker itself, the truth of the matter is that `Pickers` don't provide much out-of-the-box functionality to change their appearance. Sadly, most of the functionality gained from the fact that `Slots` extend `DataViews`, has been locked down, preventing you from passing a custom `XTemplate` for example. To truly change the look and feel of a `Picker`, you will have to venture deep into the bowels of Sencha Touch and write your own extension to the `Picker` and `Slot` class, as well as some serious CSS in addition.

On the topic of extending classes, let's see if we can get a better idea of what is achievable by extending the standard picker by taking a look at the Date Picker class.

6.2.2 Date Picker

Built on top of the standard `Picker` class, the build-in `DatePicker` consists of three `Slots`, one for year, month and day respectively. On-screen, the entire thing looks pretty much the way you would expect a `Picker` with three slots to look.

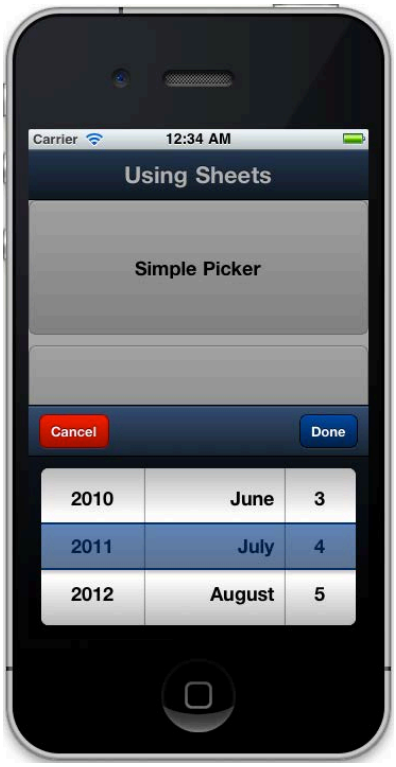

Figure 6.8 – A standard Date Picker

Where things get interesting is when taking a closer look at the configuration options and the actual code behind the `DatePicker`. To better illustrate the options available to us, let's take a look at the code for figure 6.8 above:

```
var myDatePicker = Ext.create('Ext.DatePicker', {
     yearFrom: 2010,
     yearTo  : 2025,
     slotOrder: ['year', 'month', 'day'],
     value: {
         year: 2011,
         month: 7,
         day: 4
     }
});
Ext.Viewport.add(myDatePicker).show();
```

In this particular case, we limit the dates to start in 2010 (via the `yearFrom` property) and only go up to 2025 (via the `yearTo` property). By default, the year starts in 1980 and goes

until the current year. Slots are normally sorted by month/day/year, but because we want to be friendly to our international readers, we decided to change the order. To do that, we utilize the `slotOrder` property and change it to year/month/day instead. The `DatePicker` takes care of automatically creating all the slots for us, so we don't have to worry about that part. The only thing left is to set a starting `value`. This can either be a proper JavaScript date object, or a JSON object with a `year`, `month`, and `day` property like we have above.
If you encounter one of those cases where you only need to query a user for say a year and month for example, you can simply change the slot order, and leave out the 'day' slot. This will automatically hide any slot not mentioned in the order.

CALENDAR PICKER IS MIA

For those developers coming from the ExtJS side, as of the time of this writing, the date picker is the only way to select dates, and there is no true equivalent to the ExtJS Date Picker that shows a calendar like structure split into rows and columns.

Although `Pickers` are easy to setup, and `Sheets` are very versatile in the content you can populate them with, neither one provides an easy or quick way to prompt the user for input, or relay a quick message to the user. Fortunately for us, the Sencha Touch developers have thought exactly of this use case, and given us the `MessageBox` class.

6.3 Talking to the user via a Message Box

Whether your goal is to notify the user of something that happened, prompt for input, or querying the user to make a quick yes/no decision, the `MessageBox` class is the way to go, since it provides the easiest out-of-the-box interaction with the user.

It extends the basic `Sheet` class and automatically sets a few of the configuration options associated with the `Sheet` for you. Message boxes by default always use the 'dark' `ui`, show `centered` on the screen, and 'pop' in and out for their `enterAnimation` and `exitAnimation`.

While creating a `MessageBox` largely follows the same paradigm as all the other Sencha Touch components, there are a few caveats to keep in mind that distinguish it from the rest.

First and foremost, you as a developer never have to worry about instantiating a MessageBox instance yourself. Although it is possible to do so, Sencha Touch automatically creates a singleton instance for you with the global alias Ext.Msg, thus removing the need for you to create your own before using it. The MessageBox is the only component in the framework where this happens.

What this means effectively is that instead of doing this:

```
var myMessageBox = Ext.create("Ext.MessageBox", { /*options here*/ });
myMessageBox.show();
```

You can easily do this:

```
Ext.Msg.show({ /*options here*/ });
```

The primary reason for this is that the heart of the `MessageBox` is actually in its `show` method, which allows for a rather complex configuration object to be passed to it. Before we delve into the depths of the `show` method, let's first take a look at the simplest way to show a message box via the following sample.

```
Ext.Msg.show({
   title: 'Southpark',
   msg: 'Ice Ice Timmy!'
});
```

The above code uses the message box singleton to show a simple message consisting of a `title` and `msg` (message) body. If you run the code from the console, you will notice the presence of an "ok" button, just like figure 6.9 below.

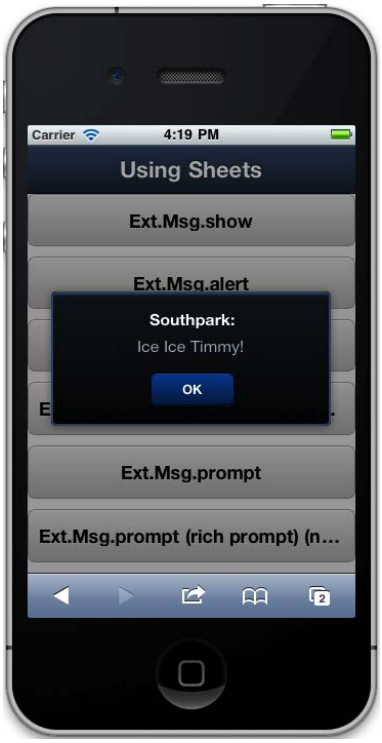

Figure 6.9 – Simple Message Box with title & message body

Even though we didn't specify it, the `MessageBox` automatically creates a `toolbar` with an "ok" button for us, to provide the user with an easy way to dismiss our message. Given that this simplistic example barely scratches the surface of what is possible, let's see if we can get a better idea of our options within the `show` method and some of the helper functions, starting with alerts.

6.3.1 Alerting Users

Alerts consist of a simple title and a message body, and are perfect to tell the user when an operation finished, an error occurred, or any similar type of message. The outcome is pretty much exactly like you've already seen in figure 6.9 above.

In the scheme of things, there are only three properties within the show method of a message box that concern themselves with showing simple text. Table 6.1 below provides details on all of them:

Table 6.1 The available show options that deal with showing simple text.

Option	Description
`title`	This sets the title of the Message Box. If this option is left blank, or omitted entirely, then no top toolbar is created. At the time of this writing, it is not possible to have a multi-line title, whether by entering long text, or forcing a line break via html tags.
`msg`	The message body to show. In the event of a prompt, this will be used as the text to show right above the prompt, similar to a label for a text field.
`modal`	A Boolean that defines whether the message box should be modal or not. This determines whether the rest of the screen should be masked or still be available for interaction.

The code to create an alert is the same as the code we used for figure 6.9 earlier. For convenience purposes, the `MessageBox` class exposes an alert function, which automatically wraps the `show` method with default values. To use the `alert` method, simply following this sample:

```
Ext.Msg.alert("Title Goes Here", "Message goes here");

Ext.Msg.alert("Title Goes Here", "Message goes here", myAwesomeFn, this);
```

The `alert` function is broken down into four parameters, the first being the title, the second being the message to show, and the third and fourth being a callback function to invoke when the message box is closed, and its respective scope. Under the hood, the `alert`

method automatically inserts an "OK" button for us, and calls the `show` method, passing along the specified parameters. In the above code, the first function call would simply show the alert with the provided `title` and `msg`. In the second call, the `myAwesomeFn` function would be invoked when the "OK" button is pressed; using the scope of whichever object called the alert. The callback function passes us three parameters. The first is the `itemId` of the button that was pressed, in this case it would be "ok". The second is the value, which only applies to prompts, and in the case of an alert is null. The last is a copy of the config object passed to the `show` method. For the most part, the callback function is used more infrequently on alerts, than any of the other types of message boxes. To see a better example of the callback being used, let's take a look at the next section, which deals with ways to prompt the user for simple yes/no type answer and then act based on those answers.

6.3.2 Prompting Users

In the world of Sencha Touch, a message box that presents the user with a set of buttons to be clicked is called a confirmation box. Typically, confirmation boxes are used in cases where the user might do something risky, and you as the developer have to protect the user from rash actions, like deleting a record, submitting a payment form, or other such thing.

In terms of coding, confirmations follow the same paradigm as the options already used for alerts in figure 6.9, with the simple addition of configuration options to define the buttons and callback function. The simplest way to invoke a confirmation message is via the `Ext.Msg.confirm` method. The syntax is exactly the same as for the `alert` method discussed earlier.

```
Ext.Msg.confirm("Are you sure?", "This will shutoff the internet!",
myAwesomeFn, this);
```

The primary difference between alert and confirm is a change in buttons. Where the alert method only shows an "ok" button, the confirmation box by default shows a "yes" and a "no" button, just like figure 6.10 below.

Figure 6.10 – Default confirmation box

For the confirmation box to be truly useful, you will have to make sure you pass a callback function so you can process which button was actually clicked. Since the confirm call above (depicted in figure 6.10) expects a callback named `myAwesomeFn`, let's see what that might look like:

```
function myAwesomeFn(buttonID, value, opts) {
    if (buttonID === "no") {
            // whew. Threat averted!
    }
    else {
            // oh noes! the internet is going down
    }
}
```

This is all that is needed to handle a confirmation box and process the response.
Wait! You might say, telling me that this can't be everything. You know for a fact you've seen message boxes with multiple buttons that were not "yes" or "no". Right you are. The yes/no combination is simply the default when using the `Ext.Msg.confirm` helper function.

To define your own buttons, you would use the show method and pass in your own `buttons` configuration, just like this:

```
Ext.Msg.show({
     title: 'Are you sure?',
     msg: 'This will shutoff the internet!',
     buttons: Ext.MessageBox.OKCANCEL,
     fn: myOtherFn
});
```

This would show the same confirmation prompt as figure 6.10, but the buttons would be ok/cancel, instead of yes/no. To better understand the different options related to confirmation dialogs, and how to define custom buttons, take a look at Table 6.2.

Table 6.2 Configuration options related to buttons, handlers, and scoping of functions.

Option	Description

`buttons`	This option defines which button(s) to show in the message box. Out of the box, there are four single buttons defined:
	▪ `Ext.MessageBox.OK`
	  ▪ Shows a simple "ok" button
	▪ `Ext.MessageBox.CANCEL`
	  ▪ Shows a simple "cancel" button
	▪ `Ext.MessageBox.YES`
	  ▪ Shows a simple "yes" button
	▪ `Ext.MessageBox.NO`
	  ▪ Shows a simple "no" button
	
	In addition to these four buttons, there are three button sets provided as well.
	▪ `Ext.MessageBox.OKCANCEL`
	  ▪ Shows a ok/cancel button combination
	▪ `Ext.MessageBox.YESNO`
	  ▪ Shows a yes/no button combination
	▪ `Ext.MessageBox.YESNOCANCEL`
	  ▪ Shows a yes/no/cancel button combination.

`fn`	This option allows you to define a callback function that is invoked whenever one of the buttons from the button configuration is

	pressed. By default, buttons provide their own scope, which means the callback would not necessarily have access to the rest of your application. To alleviate this, you can use the scope parameter.
scope	This parameter allows you to change the scope of the callback function to provide easy access to whichever level of code you need within the callback.

One often overlooked feature in regards to confirmation dialogs is the option to define completely custom buttons, which you can do by simply setting the `buttons` configuration of the dialog to an object structured like either one of the following samples:

```
Ext.Msg.show({
     title: 'Are you sure?',
     msg: 'This will shutoff the internet!',
     buttons: {
        text: 'go!',
        ui: 'action',
        itemId: 'go'
     },
     fn: myOtherFn
});
```

or for multiple buttons:

```
Ext.Msg.show({
     title: 'Are you sure?',
     msg: 'This will shutoff the internet!',
     buttons: [
        {
            text: 'go!',
            ui: 'action',
            itemId: 'go'
        }, {
            text: 'umm, no',
            itemId: 'getmeout'
        }
     ],
     fn: myOtherFn
});
```

With `alert` and `confirm` out of the way, the only mystery left to us is how to handle user input in a message box. For that, we will explore the `prompt` method in the next section.

6.3.3 Requesting input from users

Prompts provide a simple way to ask the user for a single-line or multi-line of text. Anything heavier than that, and you will have to look into building a form, something covered in chapter 7. Prompts build on top of the `alert` & `confirm` functionality we already covered in the previous section. The title is retained, and the message body is appropriated as the label for the input field. Instead of a yes/no button like the confirm box has, prompts use an ok/cancel button combination by default. Put together, this would look like figure 6.11.

Figure 6.11 – Simple Message box with Input Prompt

The default `prompt` uses a single line input field, with `autocomplete`, `autocapitalize`, and `autocorrect` turned off. Just like the other helper functions, prompt simply calls the `show` method of the Message Box, setting which `buttons` to use for you. Showing a `prompt` follows almost the same syntax as the `alert` and `confirm` dialog showcased before.

```
Ext.Msg.prompt("Sorry dude!", "What is your name again?", handleMyInput);
```

This represents the standard call for a prompt. Notice, I said that it almost follows the same syntax. The major difference is the addition of extra parameters to set whether the input field should be multiline, contain an initial value, as well as a configuration object to turn on/off things like autocomplete or set the maximum length of the field. Showing a multiline prompt looks like this:

```
Ext.Msg.prompt(
     "Sorry dude!",
     "What is your name again?",
     handleMyInput,
     this,
     true,
     "Anthony",
     { maxlength: 180, autocapitalize: true }
);
```

The first two parameters are the title and message body, while parameters three and four are the callback function and its scope. Parameter five sets the prompt to be multiline. Setting this to true means the input field will utilize the default height of 75 pixels. We could alternatively provide a number instead of true, and the supplied number becomes the height of the input field. The next parameter is the initial value we want the input field to have, while the last parameter represents the configuration for the input field. Fully rendered, the prompt looks like figure 6.12.

Figure 6.12 – A multiline prompt.

That's all there really is to prompts. The main thing to keep in mind is that you're not bound to use the prompt helper function. You can always call the Ext.Msg.show method directly and pass a configuration object with your desired combination of options. To recap the options related to text input, take a look at table 6.3

Table 6.3 The available configuration options for prompting users for input.

Option	Description
`multiLine`	This option can either be a boolean or a number. In the event it is set to "true", the input box will be multiline and use the default 75 pixel height. In the even you provide a number, the number specified will be used as the height for the input.
`prompt`	This option takes a configuration object that allows you to specify properties for the input field. The options include any of the available options for a textfield instance, which are:

	autocomplete, autocapitalize, autocorrect, maxlength, autofocus, and placeholder. All of these have defaults that are automatically set, and for the most part you wouldn't need to bother with most of these.
`value`	Automatically populates the input box with the defined value. This is only useful when you are prompting the user to provide input.

If you happen to look through the documentation, you might notice the `show` method specifies `cls`, `height`, `width`, and `defaultTextHeight` as valid options to be passed. The simple truth is, as of this writing these parameters do not accurately work, as the `show` method simply ignores them. To utilize any of these options, you will have to instantiate your own `Ext.MessageBox` instance and pass this option into the config of the `MessageBox` (instead of the `show` method), and then call the `show` method of your custom `MessageBox` instance. Using the provided `Ext.Msg` Singleton will not work with these options.

There you have it ladies and gentlemen, message boxes in a nutshell! A round of applause, please.

6.4 Summary

It feels like we're finally starting to get somewhere. Our newfound knowledge of how to overlay information onto the current screen with `Sheets`, and how to notify and prompt the user for information via `MessageBoxes`, or presenting the user with an abundance of choice through the use of `Pickers` should help make any app more intuitive and user friendly.

Together with the past several chapters, we are starting to amass an impressive list of tools in our belt, ranging from managing components, to using panels & tabs, to docking items, to harassing the user with those pesky popup messages. Although that is an impressive list, you might have the same thing going through your mind as I do: "what we've learned so far is great, but I have all this data on a remote server, and I don't know what to do with it!"; and you're absolutely correct. Without further ado, the next chapter will introduce you to the exciting world of Data Stores, Lists, and the ever so powerful Data View.

7 Data Stores and Views

This chapter covers

- How data stores work with models, proxies and readers
- Looking at how to use `DataView`
- Exploring `List` and the capabilities it has
- Digging into `NestedList` to explore how it works

We've learned quite a lot thus far learning how to work with `Panel` and `Sheet` and can build a great looking application. However, the application we can build so far is static and you may have requirements that you need to consume remote data. Say you are tasked with the task to build a mobile viewer so that anyone in your client's company can search a global address book so they can find contact information while they are traveling. You likely would want some sort of list that will load the contact list from a remote server. Might sound complicated to do and it is but Sencha Touch makes it very simple for you to accomplish this.

In this chapter we will explore `Ext.data.Store` and what classes it uses to be able to send a request to a server and read the response. We will also look at displaying this data in `Ext.dataview.DataView`, `Ext.dataview.List` and `Ext.dataview.NestedList`. Each of these three widgets builds on top of each other so we will talk about when to use one over the other.

Before we jump into using the widgets, we need to learn about data stores. A `List` cannot display anything without a data store so we will begin with an overview of what a data store is and how it works.

7.1 Examining Data Stores

`Ext.data.Store` is one of the most commonly used classes in Sencha Touch. From the perspective of someone just learning Sencha Touch it can be quite a daunting task to understand how the `Store` works and consumes remote data. Don't worry, we are going to

take it one step at a time and build our knowledge with how Stores send a request to a server and decode that data so that the Store can properly consume that data.

7.1.1 The anatomy of data Stores

If you are familiar with databases then an easy way to think of a Store is like a table in a database; it has fields and it has rows of data. That's the simplified version, but Store has many different classes it uses to load remote data and read that data so that it can consume data. To get data from a server side source you use a proxy to send that request to the server. The server will then respond with a format like JSON or XML and you will use a reader to decode the data allowing the Store to read that response. Looking at Figure 7.1, you can visualize how the process happens where the Store really only talks to the proxy and the proxy does the coordinating. There are some other classes in the mix but generally speaking this is a great overview of the workflow the Store uses to retrieve the remote data and work with it. To add some flexibility, you can use a Model class that can hold the fields, use a proxy and reader, build associations with other Models and even validate data. If you change a Model instance, or record, that is within a Store and you want to send that change to the server, the Store will use a writer to get the needed data and encode it for transfer by the proxy to the server. Figure 7.1 shows the main classes the store uses.

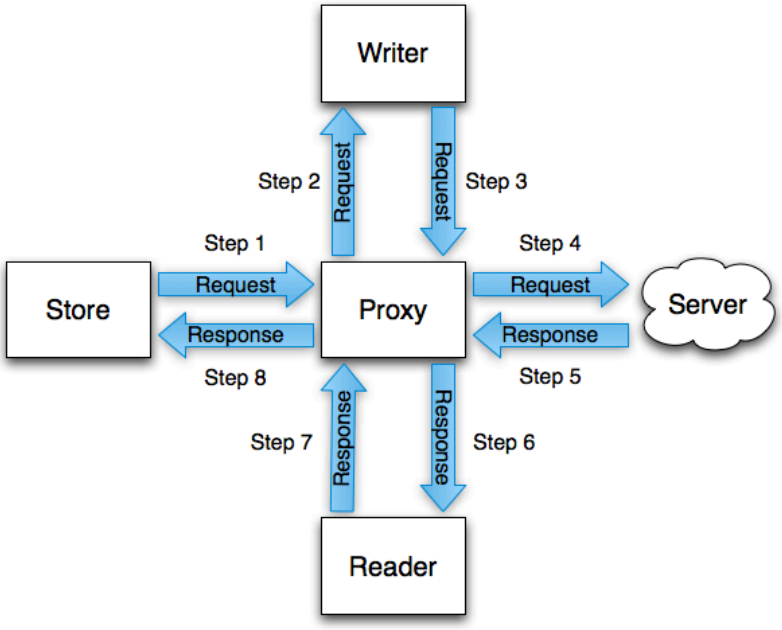

Figure 7.1 Overview of how the Store can load remote data from the server

You can see an overview about the workflow the Store has to load remote data. You can really split this up into eight steps:

Step 1 – The Store tells the Proxy that it wants to make a request.

Step 2 – The Proxy will then ask the Writer if it has anything it can add to the request the Proxy will make. This is where if you had created, edited or removed a record from the store when you do a sync request to synchronize the application with the server.

Step 3 – If the Writer has anything to add to the request, it will add data to the request. This can also be encoded into JSON or XML and other options we will discus in a little bit.

Step 4 – The Proxy now has everything it needs to make the request and so it sends the request to the server. The type of request depends on what Proxy you are using. For instance, Ajax Proxy sends an `Ajax` request where the `LocalStorage` proxy uses HTML5's localStorage.

Step 5 – The Server will do whatever it needs, gather data and/or handling the new, edited or removed records and respond. The response will come back in either JSON or XML, whatever you choose as your response.

Step 6 – The Proxy got the response from the Server but it's basically just a string, either JSON or XML with JSON being the recommended. But as is, the Store won't be able to handle this raw data. So the Proxy sends the raw data to the Reader for decoding.

Step 7 – The Reader does the decoding taking the JSON or XML and turning it into JavaScript.

Step 8 – The round trip for the request is now complete. The Proxy will create records from the JavaScript and return the records to the Store.

Now the different widgets are now able to display the data. It's good to know about `Ext.data.Operation`. This class is a heavy lifter behind the scenes doing much of the work and works in parallel with the Proxy. This class is more for advanced users, as you really are not going to deal with Operation directly. Let's slow it down and review the different classes that we just discussed in more detail.

7.1.2 Using proxies to load data

As mentioned, you use a proxy in a `Store` to load data but what does that mean? I'm sure you are familiar with Ajax and how it can send a request to a server and it returns a response. That is basically what a proxy does; sends a request to a server to get data remotely. In fact the most popular proxy class is the `Ext.data.proxy.Ajax` proxy, which uses an Ajax request to get data.

There are other proxies that you can use depending on your server requirements but they all work the same. You have `Ajax`, `Direct`, `JsonP`, `LocalStorage`, `Memory`, `Rest` and `SessionStorage` proxies and they all work the same just different was of communicating with a data source.

Table 7.1 shows some of the more used configurations of the proxies

Table 7.1 Describing the common configurations of a proxy.

Config	Description	Defaults
url	The URL of the remote data source	Empty string, ''
api	An object of urls for each CRUD action. You should only use this over the url config if you need to have different urls for each action.	{ create : undefined, read : undefined, update : undefined, destroy : undefined }
actionMethods	An object of methods to use for each CRUD action.	{ create : 'POST', read : 'GET', update : 'POST', destroy : 'POST' }
extraParams	An object of parameters to include in the request. These will always be sent but can be overridden in the params config of the load method.	Empty Object. {}
reader	A configuration object to configure the reader.	{ type : 'json' }
writer	A configuration object to configure the writer.	{ type : 'json' }

NOTE All proxies, except the default `Memory` proxy, load data asynchronously so you need to use a callback or an event to take action when a `Store` loads.

There are a couple web sites that can help you debug responses if you are returning JSON data. If you are using the `Ajax` proxy with `Json` reader then http://www.jsonlint.com is a great place to test your server so that it is returning what you are expecting and also indent the JSON so you can better visualize the response. If you are using `JsonP` proxy then http://www.jsonplint.com is the site to do your server testing and response visualization. Both sites can handle remote requests or you can paste the response to lint the response.

JSONP

JSONP is a means of loading cross-origin content without CORS (Cross-Origin Resource Sharing) to get around browser security. CORS allows you to make Ajax calls cross-origin but requires server setup. A quick reference to setting CORS up is at http://www.enable-cors.com/. However, the server has to respond with a valid JSONP response. A valid response is JSON surrounded by a callback function whose name is sent in the request. For example, if in your request you have a callback parameter with a value of `'Ext.data.JsonP.callback1'` then the response should look like: `Ext.data.JsonP.callback1({"foo": "bar"});` More can be found at http://www.jsonplint.com

7.1.3 Using readers to digest data

Like the different proxies, there are different readers and to decide which to use depends on the server response. You have three readers to use: `Array`, `Json` and `Xml` and which to use depends on what your server is going to respond with. The recommended is for your server to return JSON therefore you should use `Ext.data.reader.Json` to decode that JSON. If your server returns Xml then you should use `Ext.data.reader.Xml`.

Table 7.2 shows some important configurations of a reader

Table 7.2 Describing the common configurations of a reader.

Config	Description	Defaults
idProperty	The field name in the response to use as the id of each record.	'id'
messageProperty	The property in the top level of the response to use as the message of the response. This is useful if there was an error that the server needs to reply with the error message.	null
rootProperty	The property in the top level of the response to use as the root of the data.	Empty String. ''
successProperty	The property in the top level of the response to use as the success signal.	'success'
totalProperty	The property in the top level of the response to use as the number of total results. This number is used for paging and tells the store how many total results not how many are in the returned page.	'total'

7.1.4 Understanding Models

`Models` are the rows of data within the `Store` and must have the fields defined. In a real world application you should have your code organized so that you always have a `Model` defined for a `Store` to use and shouldn't really define fields on the `Store`; only if you are creating something simple should you define fields on the `Store` and not the `Model`, even then it can be argued that you should use a `Model`.

Common convention is to refer to `Model` instances as records also so if you see the word record, just think `Model` instead. The reason for this is legacy. Ext JS versions 1 through 3 used records, but in Sencha Touch 1 and Ext JS 4 the whole data package was updated and we now use `Model` instead of record. Old habits die hard for those of us that have been in the Sencha environment for many years.

Models can also live outside of a Store so that you could load a single row of data. In order to do this you could define the proxy and reader on the Model. If you do define the proxy and reader on the Model and the Store does not have a proxy and reader defined, the Store will use the proxy and reader from the Model to use. If you have a proxy and reader defined on both the Store and Model, the proxy and reader on the Store will be used when loading the Store.

7.1.5 Writer to sync

The last bit to talk about to complete the Store ecosystem is the writer. A writer's job is to include new, edited or destroyed records in the sync request from the Store. You execute a sync request by executing the sync method on the Store and if there are any dirty records they will be included in the request. A dirty record describes a record that is new, edited or destroyed. You will also see the term phantom record which is a new record on the client side that has not yet been synced with the server side.

Like the proxies and readers, there are different writers depending on your situation. If you are using JSON then you should use the Json writer; likewise if you are using XML then you should use the Xml writer. Table 7.3 shows the common configurations of the writers but each writer has its own specialized configs for their request format so there are some others.

Table 7.3 Describing the common configurations of a reader.

Config	Description	Defaults
encode	Only for Json writer, set to true to encode the request and be placed in the request body or if rootProperty is set as the value of that property.	false
writeAllFields	For both Json and Xml writers, set to true to send all field values, else only the ones that were changed.	true
rootProperty	Only for Json writer, the property in the top level of the request to send the encoded data.	'records'
documentRoot	Only for Xml writer, the property to be used as the root of the XML document.	'xmlData'

7.1.6 Simple Store example

The example in Listing 7.1 shows a simple example of how to setup a local Store. It's called local as the data is specified locally, in this case inline, whereas a remote Store would load remote data via a proxy.

Listing 7.1 Simple local Store example

```
Ext.define('MyModel', {                                          // 1
    extend : 'Ext.data.Model',

    config : {
        fields : [                                               // 2
            'name',
            { name : 'twitter', type : 'string' }                // 3
        ]
    }
});
var store = new Ext.data.Store({                                 // 4
    model : 'MyModel',                                           // 5
    data  : [                                                    // 6
        { name : 'Mitchell Simoens', twitter : 'msims84'  },
        { name : 'Jay Garcia',       twitter : '_jdg'     },
        { name : 'Anthony De Moss',  twitter : 'ademoss1' }
    ]
});
```

{1} Use Ext.define to create the Model class
{2} Specify the fields array
{3} Use a config object for Ext.data.Field
{4} Instantiate the Store
{5} Bind the Model to the Store
{6} Use inline data

As you can see from Listing 7.1, implementing a local `Store` with inline data is very simple. I know we are using something new, `Ext.define` **#1**, here without explaining it; we will cover it more in Chapter 10 but it simple creates a new class definition and in this case a `Model` class to be used for the `Model` instances. A `Model` has to have fields so we use the `fields` configuration **#2** to specify an array of fields. You can use a string or a configuration object to specify your fields, if you specify a string it will be used as the name and no conversion will happen. However, if you specify a configuration object **#3**, then it will be used to create the field instance. If you need to do value conversion you can specify the type configuration; reference `Ext.data.Field` for more configurations that you may need.

Now we get into actually creating the `Store` **#4**. We use the `Model` configuration **#5** to tell the `Store` which `Model` class definition to use. To finish off a simple local `Store` we give it inline data **#6**, which is an array of objects. Notice the properties name and twitter in the data objects will map to the fields in the `MyModel Model` class definition. With this simple code your `Store` now has 3 records (`Model` instances) and is ready for a view to display this data.

Now let's look at implementing a proxy and reader so we can have a remote `Store`. Configuration is pretty simple but we will highlight some of the main areas in Listing 7.2. We will reuse the `MyModel Model` we created from Listing 7.1.

Listing 7.2 Simple remote Store example

```
var store = new Ext.data.Store({
```

```
    model    : 'MyModel',
    autoLoad : true,                                                    // 1
    proxy    : {                                                        // 2
        type   : 'ajax',                                                // 3
        url    : 'authors.php',                                         // 4
        reader : {                                                      // 5
            type          : 'json',                                     // 6
            rootProperty  : 'authors',                                  // 7
            totalProperty : 'totalNumber'                               // 8
        }
    }
});
```

{1} Use the autoLoad config to automatically load the Store
{2} Specify the proxy config on the Store
{3} Use the type config to specify the type of proxy
{4} Specify the url to load the data from
{5} Specify the reader config on the proxy
{6} Use the Json reader
{7} Specify the root of the data
{8} Change the total property

Remembering that we are reusing the `MyModel` `Model` class from Listing 7.1 and picturing the anatomy of the `Store` from Figure 7.1, this remote `Store` is very easy to understand how it works. By default, the `Store` will not load automatically so we can specify the `autoLoad` configuration **#1** to true to make the `Store` to load right away. We could also execute the load method after the `Store` was created to have control of when the `Store` loads. In order to load we specify the `proxy` configuration **#2** to tell the `Store` to initialize a proxy. We are going to use `Ajax` to load remote data so we specify the `type` configuration **#3** and `url` configuration **#4** so we can use the `Ajax` proxy. This may throw you but looking back at Figure 7.1, we can see the proxy talks to the reader not the `Store` so we put the reader configuration **#5** in the `proxy` configuration object not the `Store`. To configure the reader we tell it to use the `Json` reader with the `type` configuration **#6** and also use the `rootProperty` **#7** and `totalProperty` **#8** to properly map where the data is and the total number of records in the database.

This `Store` configuration would expect the following response from authors.php

```
{
    "success": true,
    "totalNumber": 3,
    "authors": [
        {
            "name": "Mitchell Simoens",
            "twitter": "msims84"
        },
        {
            "name": "Jay Garcia",
            "twitter": "_jdg"
        },
        {
            "name": "Anthony De Moss",
```

```
            "twitter": "ademoss1"
        }
    ]
}
```

We now know how to configure both local and remote Stores and what a sample response would look like for the remote Store. Time for the fun part: learning and creating a few different views to use the Store to display data. First, let's look at the DataView.

7.2 Implementing DataView

We now know how to get data from a data source but now we need to learn how we can display this data. For this you can use the Ext.dataview.DataView widget that take the records from a Store and display them with an XTemplate. You can get as advanced as you would like or keep it very simple.

7.2.1 How DataViews work

Just like when we walked through the different classes for the Store, we need to know what different classes the DataView uses. As we learned from Chapter 3 that there is a Component Model, one component extends another component in order to give stability and standard behavior. DataView extends Container and each record in the Store gets rendered as an item to the DataView; so now you have items within the parent container (DataView). Figure 7.2 shows a visualization that DataView extends Container and gets the records from the Store to render its items. What does that mean? Well, DataView gets all the records within the Store, loops through that array to generate HTML using the record data and the template you defined in the itemTpl configuration to generate HTML for each row. What is that itemTpl configuration I mentioned? It's an Ext.XTemplate configuration usually a string that generates HTML when you pass it data.

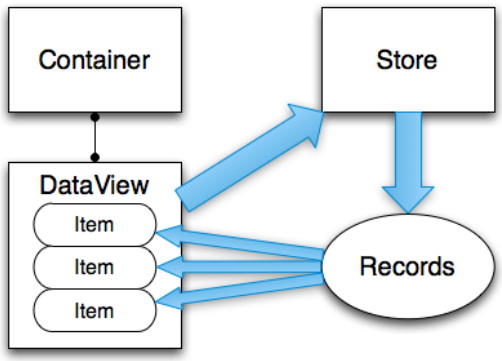

Figure 7.2 Showing the workflow of how DataView gets the records from the Store to render its items

7.2.2 Walking through XTemplate

What is an XTemplate? It's really its own language but think of it as a template but on steroids. It can map data in a simple form or it can get more advanced where you can use loop through an array in your data, use template methods to have some logic to change or format what is to be displayed and even execute JavaScript. XTemplate is intimidating, I will give you that but learning it piece by piece you can learn how to take advantage of this class very easily.

Let's look at XTemplate's simplest form where you can simply map data to display data-driven HTML:

Listing 7.3 XTemplate example

```
var template = new Ext.XTemplate(                                    // 1
    'Book: ',                                                        // 2
    '{data}'
);

var html = template.apply({                                          // 3
    data : 'Sencha Touch in Action'                                  // 4
});

Ext.getBody().setHtml(html);                                         // 5
```

{1} Create the XTemplate
{2} Specify the strings to be used
{3} Use the apply() method
{4} Specify the data property
{5} Display the compiled HTML

That was simple! First we created an XTemplate **#1** and passed in a couple parameters. In this case we passed in two strings **#2** but you can pass in a couple different things. You can pass in one string or a number of strings that will get combined into one. You can also pass in an array that will get joined into a single string. We then used the apply method **#3** on the XTemplate to apply the object of properties and values that will get mapped. This object has a data property **#4** that will get mapped to the {data} string within the string. Lastly we want to display the HTML that was just created so we use the setHtml method **#5** on Ext.getBody(), which is just the document.body. The results of Listing 7.3 can be seen in Figure 7.3.

Figure 7.3 Results of the simple XTemplate

We are getting more and more advanced and it's really not that difficult to take advantage of the power behind `XTemplate`. The next thing we can learn is looping through an array so we can display data that may be in an array form. For this we use a for loop like we would in JavaScript but we don't have to worry about setting an iteration variable or specifying a length so the for loop can know how to loop through an array, `XTemplate` will take care of this for you:

Listing 7.4 XTemplate array looping

```
var template = new Ext.XTemplate(
        'Book: ',
        '{data}',
        '<tpl for="authors">',                                    // 1
            '<p style="padding-left: 2em;">{name}</p>',           // 2
        '</tpl>'
);

var html = template.apply({
    data    : 'Sencha Touch in Action',
    authors : [                                                   // 3
        {
            name : 'Mitchell Simoens'
        },
        {
```

```
            name : 'Jay Garcia'
        },
        {
            name : 'Anthony De Moss'
        }
    ]
});

Ext.getBody().setHtml(html);
```

{1} Use the <tpl> tag to loop
{2} Map to the name property
{3} Specify the authors array

This example builds on Listing 7.3 in easily show how to loop through data. When `XTemplate` parses the string you pass it, if it sees the `<tpl>` tag **#1** it knows that it has something to do. In this case we use the `for` attribute which tells it that it needs to loop through the `authors` property in the data passed in. Each iteration through the `authors` array that object will get applied to the string within the `<tpl>` and `</tpl>` so we can use the `{name}` **#2** to map to the `name` property of the objects within the `authors` array **#3**. It can be hard to visualize sometimes what this will produce, Figure 7.4 will show the output, notice the indented lines is what is in the array that we looped through.

Figure 7.4 Results of looping through an array within an XTemplate

This next example can show two ways you can format the data getting mapped. Say we are expecting an object with a date and we want to format it. Listing 7.5 will show how we can use `Ext.util.Format` and also how we can use member functions to return a formatted date string:

Listing 7.5 XTemplate array looping

```
var template = new Ext.XTemplate(
        '<p>Ext.util.Format: ',
        '{dateVal:date("Y-m-d")}</p>',                              // 1
        '<p>Member Function: ',
        '{[this.formatDate(values.dateVal)]}</p>',                  // 2
        {
            formatDate : function(date) {                           // 3
                return Ext.util.Format.date(date, "Y-m-d");
            }
        }
);

var html = template.apply({
    dateVal : new Date()
});

Ext.getBody().setHtml(html);
```

{1} Shortcut to the Ext.util.Format.date method
{2} Execute a member function
{3} Specify the member function

In this example we see two different ways to format a date but as an example of how to map to `Ext.util.Format` and use member functions. When you see `{dateVal:date("Y-m-d")}` it knows that you want to execute `Ext.util.Format.date` passing in the value of the mapped data in this case is the `dateVal` property and also pass in the format string **#1**. This is actually the same as doing it in the member function, which we tell it to execute via the square brackets **#2**. The scope is that of the `XTemplate` and the member functions specified in the object (in the same place you specify the strings) are placed on the `XTemplate`. We have to get the data point from the values object which when you apply data to the template the `formatDate` member function **#3** will then execute. Do note that you should always return something in the member function. Figure 7.5 shows the output of Listing 7.5 using the shortcut method **#2** and the member function **#3**.

Figure 7.5 Shows the output of two ways to execute functions within XTemplate

XTemplate isn't meant to replace JavaScript just make it easier to give control over what is to be displayed by allowing JavaScript to be executed from within XTemplate. Here is how we execute JavaScript from within XTemplate:

Listing 7.6 XTemplate array looping

```
window.STIA = {};                                                    // 1

STIA.formatDate = function(date, format) {                           // 2
    return Ext.util.Format.date(date, format);
};

var template = new Ext.XTemplate(
        '<p>Javascript execution: ',
        '{[STIA.formatDate(values.dateVal, "Y-m-d")]}</p>'           // 3
);

var html = template.apply({
    dateVal : new Date()
});

Ext.getBody().setHtml(html);
```

{1} Create the STIA namespace
{2} Create the function to execute

{3} Execute the JavaScript function

This is very similar to executing member functions. First we specify the function that we want to execute, this case it's a `formatDate` method on the `STIA` namespace **#2** that we created **#1**. Then just like we did with the member function, we execute the method within the square brackets **#3**. The results of Listing 7.6 can be seen in Figure 7.6.

Figure 7.6 Shows the output when executing your own JavaScript within XTemplate

We have walked through some very useful features of `XTemplate` to display data-driven HTML to view. You may be thinking `XTemplate` is great and it is but it has a downside. With everything you do in a mobile app, you have to think performance. If you overuse `XTemplate`, then rendering the `DataView` will feel very slow because all the code within the `XTemplate` will get executed for each record in the `Store` so if you have 50 records in the `Store`, your code will execute 50 times. You need to ensure your `XTemplate` is as optimized as you can get it while still accomplishing what you need to accomplish.

7.2.3 Implementing our first DataView

We have learned about the `Store` and its proxy, reader and `Model`; we have also learned about how the `DataView` works and what functionality the `XTemplate` has but how do we put all that together to make a fully working `DataView`? That's exactly what we are about to walk through and it's easier than you think.

We will use the store from Listing 7.1 to already have the data present for our DataView to display; we will use the remote Store in Listing 7.2 when we try out the List. First let's look at Listing 7.7 to see how we can implement our first DataView.

Listing 7.7 Simple DataView

```
var dataview = new Ext.dataview.DataView({                          // 1
    fullscreen : true,                                              // 2
    store      : store,                                             // 3
    itemTpl    : [                                                  // 4
        '{name} ',
        '<a href="http://www.twitter.com/{twitter}" target="_blank">',
            '{twitter}',
        '</a>'
    ]
});
```

{1} Create the DataView
{2} Add to Ext.Viewport
{3} Specify the store to use
{4} Use XTemplate to display the data

The first step is to instantiate the DataView in this case via direct instantiation **#1**. We use the fullscreen configuration **#2** to add the DataView to Ext.Viewport in order to render the DataView to the DOM so we can see the data. We use the Store from Listing 7.1 with inline data by using the store configuration **#3**. Lastly we need to use XTemplate to get the data from the store to display using the itemTpl **#4**. Since the Store has data in it, DataView will take that data and apply it to the XTemplate we specified in the itemTpl configuration, which will result in Figure 7.7. You can see we use {name} and {twitter} to tell XTemplate where to display the data from the Store.

Figure 7.7 Results from Listing 7.7 showing inline data from the Store using DataView

One thing I would like to say about `itemTpl` is that it can accept a few different types of values. In Listing 7.7 we specified an array of strings but we could have specified a string or an actual `XTemplate` instance. It is common to just specify a string or an array of strings; I used an array only to fit it on the page of this book.

We just learned the first widget, `DataView`, which can display data from a `Store`. Let's build on a subclass of `DataView` the `List` where we will learn about some of the differences and features that `List` has.

7.3 Advanced features with List

So far in this chapter we have learned about how to get data with the `Store` and display the data in the `DataView` using `XTemplate`. In this section we are going to take a look at the `Ext.dataivew.List` component, how it differs and use the extra functionality it has.

7.3.1 How List differs from DataView

`List` extends `DataView` so you get all the functionality that we have learned thus far in this chapter like getting data from a `Store` and displaying it with an `XTemplate`. There are four things that `List` can do that `DataView` cannot do:

1. CSS – The `List` has some CSS for it that the `DataView` does not have.

2. IndexBar – This is a vertically stacked alphabet that when tapped on will jump to the group of records. You should only use this when your List is grouped
3. Group Data – Each of your records can be grouped and get a header to specify the group.
4. Disclosure Icon – An icon usually shown to the right on each row that signifies an action can take place when you tap on that icon.

Taking it slow, let's look at how to create a simple List. We will use the same Store and itemTpl as we did with Listing 7.7 so that we can compare the results of the simple List versus the simple DataView.

Listing 7.8 Simple List

```
var list = new Ext.dataview.List({                                    // 1
    fullscreen : true,
    store      : store,
    itemTpl    : [
        '{name} ',
        '<a href="http://www.twitter.com/{twitter}" target="_blank">',
            '{twitter}',
        '</a>'
    ]
});
```

{1} Create the List

I have to admit something; I committed the common copy/paste rule that you shouldn't do. All I did with Listing 7.7 is change the first line **#1**. For this simple List the configurations are exactly the same as the simple DataView in Listing 7.7. Let's look at Figure 7.8 to see the differences.

Figure 7.8 Differences of a simple DataView and List

On the left we have the `DataView` and the right we have the `List`. The only difference is visual really here as the List has some CSS to make the rows more distinguished. Another difference is if you tap on a row in each, the `List` will display the row with a blue background and white text to signal it has been selected whereas the `DataView` doesn't have this CSS. You would need to handle that yourself in your CSS.

7.3.2 CSS differences between List and DataView

As you can see in Figure 7.8, the `List` has a different look than the `DataView`. The DOM (Document Object Model), or the elements that you see, structure is pretty much the same so why do they look different? That is because the CSS that is bundled with Sencha Touch has styling for the `List` where the `DataView` is meant to allow the developer to style it how he or she needs without overriding styles. Another CSS difference is when a row is selected. Both the `DataView` and `List` can allow each row to be selected and if you look at the CSS classes on the row element you will notice it gets a CSS class name added, usually `x-item-selected`. When you select a row in the `DataView`, nothing visually will happen but if you select a row in the `List` you will notice that row will get a blue gradient background and the text will be white to show the selection. You could specify your own CSS to handle selection but with `List` you will have to keep in mind that there is already CSS styling that you may have to override, as with `DataView` there is none.

CSS styling is the first of the differences that `List` has over `DataView`. The other 3 are going to be grouped into one section with the example as the next section as they are just more advanced features. This section will be exciting to see more visual changes happen but first we must do our homework before playing.

7.3.3 Advanced features for List

We have three more differences the `List` has over its superclass `DataView` has to talk about which are more features. Since they are features and to save time we can discus each and then see a single example. Don't worry, when we see the example we will of course walk through the code so you can see what lines of code is doing what to accomplish the what we see in Figure 7.9.

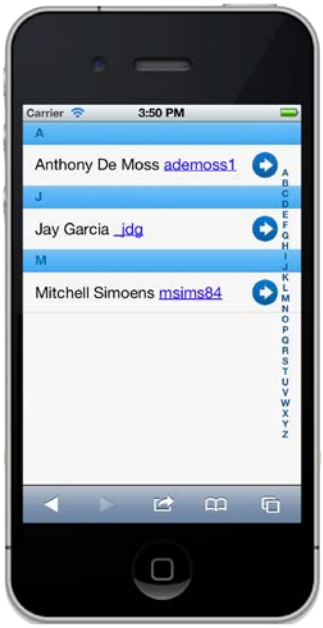

Figure 7.9 The List showing IndexBar, groups and disclosure icons

IndexBar - Like many native apps that list data, they usually have what is called an `IndexBar`. Using Figure 7.9, the `IndexBar` is on the right side of the `List`. This `IndexBar` is a vertical stack of the alphabet that is docked to the right side of the `List` and when tapped allows your `List` to jump down to the group that matches the letter tapped. You will usually only see the `IndexBar` in conjunction when a `List` is grouped or sorted. Do also note that it will show all letters despite in Figure 7.9 only having 3 groups.

Grouped Data – Say you have a list of people both with a first name and last name as people usually do. We could present the user using the application a long list of people and maybe even sort this list by last name. We can imagine what this list will look like but scrolling a long list searching for a particular person our users may get lost. What we could provide our users is a header over a group of records; for instance, all last names that begin with A will be under the A header and last names that begin with X will be under the X header. Looking at Figure 7.9, we can see a short `List` grouped with the blue headers over each group. If we were to scroll and have more rows, the header will be docked to the top until the next header reaches the top where that header will then be docked to the top. We do need to know that the `Store` will need to be configured to be able to group (which will also sort based on the grouping) so the data being provided to the `List` is in order making it easy for the `List` to display the data. We will look at this `Store` modification when we look at the example shortly.

Disclosure icon – Still looking at Figure 7.9, notice the icons on each row floating to the right? These icons, called a disclosure icon, to the right isn't just cosmetic but will change and add behavior when tapped on. Normally when you tap on a row it will select but if you tap on the disclosure icon it will not select the row but will fire the `disclosure` event on the list. The `onItemDisclosure` configuration on the `List` can be set to a Boolean or a function. If the `onItemDisclosure` configuration is specified as a function, when you tap on the disclosure icon it will fire the `disclosure` event and also will execute the function giving you two ways to handle the disclosure icon tapping. In your application I would stick with using the `disclosure` event but in some cases it may be easier just to handle the tap in the `onItemDisclosure` configuration function. In this example we will use both the `disclosure` event and specify the `onItemDisclosure` configuration as the function but do know that you could just specify the `onItemDisclosure` configuration as true and use the `disclosure` event.

We just went over the three new features the `List` has that the `DataView` doesn't have. We also learned that the `List` has some CSS styling that the `DataView` does not have. Can you imagine what all these differences look like? No? Well, let's build an example and see what the `List` will look like with all these differences working together.

7.3.4 Example of IndexBar, grouping and disclosures

In Listing 7.9 we enabled the `IndexBar`, grouping and disclosures to show how configurable the `List` is and how simple it is to accomplish. We will also use the remote `Store` from Listing 7.2 with a little modification for grouping so we can see what happens when the List has to wait for data to load so it can display. You can add the following grouper configuration to the Store to enable grouping:

```
grouper  : {
    groupFn : function(record) {
        return record.get('name')[0];
    }
```

```
}
```

What this does is for each record; it gets the value from the name property and returns the first letter. This now makes all records be grouped on the first letter of the name property. Now let's use this and others for an advanced `List` in Listing 7.9:

Listing 7.9 Advanced List

```
var list = new Ext.dataview.List({
    fullscreen       : true,
    store            : store,
    indexBar         : true,                                    // 1
    grouped          : true,                                    // 2
    onItemDisclosure : true,                                    // 3
    itemTpl          : [
        '{name} ',
        '<a href="http://www.twitter.com/{twitter}" target="_blank">',
            '{twitter}',
        '</a>'
    ],
    listeners        : {
        disclose : function(list, record) {                     // 4
            Ext.Msg.alert(
                'Author Tap',
                'You tapped on ' + record.get('name')
            );
        }
    }
});
```

{1} Config to use the IndexBar
{2} Enabled List grouping
{3} Show disclosure icons
{4} Listen to the disclose event

Taking from section 7.3.2, we set the `indexBar` configuration **#1** to true. We enable grouping using the `grouped` configuration **#2** so now we have the headers to distinguish the different groups. To display the disclosure icons we set the `onItemDisclosure` configuration **#3** to true and listen for the `disclose` event **#4** to show an `Ext.Msg.alert` when we tap on one of the disclosure icons.

Figure 7.10 Shows the List loading, with the three features and handling the disclose event.

While the `List`, `DataView` also, is loading; by default it will show an instance of `Ext.LoadMask`. You can configure this by using the `loadingText` configuration of `List` or `DataView`. The middle image is what the `List` looks like with the `IndexBar`, grouping and disclosure icons all enabled on the `List`. The `IndexBar` is the vertical alphabet on the right and if you had a long `List` and you would tap on a letter, the `List` would scroll down to the beginning of that group. If a group doesn't exist for that letter it will go to the letter before it. To show that the `List` is grouped, the `List` displays the headers for each group with `groupFn` value in the bright blue header. You see the blue circle with a white arrow in it? That's the disclosure icon and if you were to tap it, Listing 7.9 will display an `Ext.Msg.alert` when the `disclose` event fires.

Of course you can use each of these three features of `List` separately of each other so you can have just disclosure icons or grouped with or without the `IndexBar` but this example shows a decently advanced example of what the `List` is capable of out of the box.

Now we have seen the features of the `List` and things look really good! We can build some fantastic `Lists` for our users that are visually rich. Next we are going to talk about `NestedList`, which is an extension of `Container` that uses child `Lists` to display hierarchical data in a `Card` layout.

7.4 Displaying hierarchical data with NestedList

If you have flat data, DataView or List is what you should use but if you have hierarchical data then they are not equipped to handle that data. They will show the first level, also called the root level, of data but not any of the child data levels. Not to leave you out in the cold, Sencha Touch has a component equipped to handle hierarchical data called Nestedlist. As the name suggests, the NestedList component will show a List for each nested level of data. NestedList extends Container and uses Card layout. Each item is a List and the number of items depends on the number of levels in your hierarchical data. NestedList also has a top docked TitleBar to show a back Button when not on the root level allowing you to traverse backwards. This TitleBar will also show the title text usually the same text that is shown on the row you tapped on.

To get the data you don't just use a regular Store like we used for DataView and List; Store also doesn't support hierarchical data. For this we have to use a TreeStore.

7.4.1 Understanding the hierarchical data

Before we go diving into the NestedList and TreeStore we need to look at what the hierarchical data should look like. Hierarchical data is nested data with each record is known as a node. The data becomes hierarchical when a node contains child nodes and those can container child nodes and so on. A node that doesn't have any children is known as a leaf. I'm sure you are familiar with the file system on a computer where folders can have folders and files. A node that has children is like a folder and a leaf is like a file.

Here is what a simple JSON sample of hierarchical data would look like:

```
{
    "children" : [
        {
            "text"     : "Mitchell Simoens",
            "children" : [
                {
                    "text" : "@msims84",
                    "leaf" : true
                },
                {
                    "text" : "http://www.linkedin.com/in/mitchellsimoens",
                    "leaf" : true
                }
            ]
        },
        {
            "text"     : "Jay Garcia",
            "children" : [
                {
                    "text" : "@_jdg",
                    "leaf" : true
                },
                {
                    "text" : "http://www.linkedin.com/in/tdginnovations",
                    "leaf" : true
                }
```

```
            ]
        },
        {
            "text"     : "Anthony De Moss",
            "children" : [
                {
                    "text" : "@ademoss1",
                    "leaf" : true
                },
                {
                    "text" : "http://www.linkedin.com/in/ademoss",
                    "leaf" : true
                }
            ]
        }
    ]
}
```

Looking at the JSON above, you can see the hierarchy. The top level is the root of the data, which has a single property, named `children`. Each level will have a `children` property to specify what children that item has unless that item has the `leaf` property. It is assumed each item has children unless the `leaf` property equates to `true` meaning that it does not and will not have children; it's the end of the road. For some text to display, each item, leaf or not, will have a `text` property which will be displayed.

7.4.2 Using TreeStore

Like I mentioned at the beginning of section 7.4, we need to use `TreeStore`, not a regular `Store` to consume hierarchical data. There isn't a difference when creating a `TreeStore` really, it's just the `TreeStore` is able to handle hierarchical data where `Store` cannot.

Using the same JSON we looked at in section 7.4.1, Listing 7.10 is a sample `TreeStore` with the associated `Model`:

Listing 7.10 Sample TreeStore

```
Ext.define('Author', {                                          // 1
    extend : 'Ext.data.Model',

    config : {
        fields : [
            'text',                                             // 2
            'link'                                              // 3
        ]
    }
});

var store = new Ext.data.TreeStore({                            // 4
    model    : 'Author',
    autoLoad : true,
    proxy    : {
        type  : 'ajax',
        url   : 'authors-tree.json'                             // 5
```

```
    }
});
```

{1} Create the Author Model
{2} Specify the text field
{3} Specify additional fields
{4} Create the TreeStore
{5} Specify the JSON file

Creating a `TreeStore` is really simple. Like a regular `Store`, we need to create a `Model` class definition and in this case we are naming it `Author` **#1**. Only thing required is to specify the field named to the field in the JSON that we will be using to display, which is `text` **#2**. We also specified another field that we will use later named `link` **#3**. Now we create the `TreeStore` **#4** and give it the `model`, `autoLoad` and `proxy` configurations and within the `proxy` configuration we are going to use the `Ajax` proxy with the `url` to the `authors-tree.json` file **#5** that contains the JSON from section 7.4.1.

Run the code from Listing 7.10 and the `TreeStore` will load the JSON from the authors-tree.json file and parse the hierarchical data to be used in a `NestedList`.

7.4.3 Creating a Simple NestedList

So we can load data but that's no fun without displaying the data. As we discussed earlier, we cannot use a `DataView` or a `List` to display hierarchical data. For this, Sencha Touch has the `NestedList` component. Remembering from earlier in section 7.4, `NestedList` is just a `Container` that uses `Card` layout to display child `Lists`. Here is a simple example of how to display the JSON from section 7.4.1 using the `TreeStore` from Listing 7.6:

Listing 7.11 Simple NestedList

```
var nestedlist = new Ext.dataview.NestedList({                    // 1
    fullscreen : true,
    store      : store,                                           // 2
    title      : 'Authors'                                        // 3
});
```

{1} Create the NestedList
{2} Specify the store
{3} Specify the title of the root

Not much going on in a simple `NestedList`. Of course we create the `NestedList` **#1** and give it the `fullscreen` configuration to add it to `Ext.Viewport` so it can be displayed. We also specified the `store` **#2** for the `NestedList` to use. We also give it the `title` configuration, which is a title **#3** for the root level. The `title` configuration is actually an optional configuration but without it the `NestedList` will look a little funny, as the top docked `TitleBar` will be blank. With the `title` configuration, when you are on the root level the `TitleBar` will display this value so your user can visually see what they are looking at. Check out Figure 7.11 to see what you would see when you run Listing 7.11.

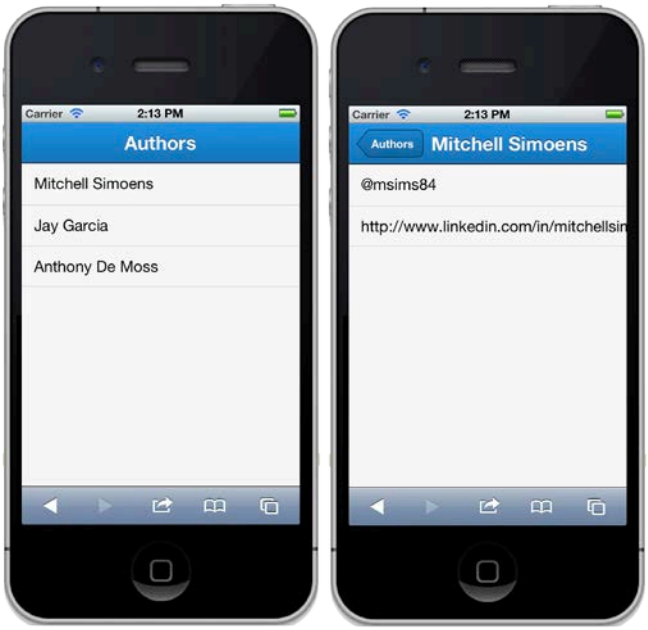

Figure 7.11 Results of Listing 7.7 showing a simple NestedList

Looking at Figure 7.11, the image on the left is the root level. You can see the title that we specified in Listing 7.11 at the top. Without it, you would just get the blue toolbar docked there, which doesn't look the best. Plus, with the `title` configuration set, we can now see that this list is the `Authors`. Imagine we tapped on the first list item `Mitchell Simoens`, it would then animate to the image on the right to show the children of the `Mitchell Simoens` list item. Do note that the top `TitleBar` now displays the same text for the title as the list item we tapped on. If we tapped on `Anthony De Moss`, it would say `Anthony De Moss` instead of `Mitchell Simoens` in the `TitleBar`. If the text were longer than can fit in the `TitleBar`, it would be cut off and have the ellipsis to give a better presentation than just letting the text run off screen. Also not in the `TitleBar` that we now have a `Button` that says we can go back to the `Authors` list. This `Button` uses the `back ui` and uses the text from what was in the `TitleBar` as the title in this case the last title was `Authors`. If we didn't specify the `title` configuration on the `NestedList`, the `Button` would display the text `Back` so the back `Button` should never be blank.

7.4.4 Showing details

So what happens when we get to the level that is a leaf? You can tap on it and it will get selected but nothing happens. Of course the reason is because it is a leaf you tapped on so there are no more children but usually you would want to show details about that leaf. A

configuration on the NestedList that we haven't covered so far is the detailCard configuration. What this detailCard configuration does is when you tap on a leaf it will create and add this detailCard to the NestedList and animate to this component. The detailCard can be anything you want it to be, another List, a Container or a form; anything you need it to be.

Let's take a look at a sample of how to work with it and then we can talk about what we see.

Listing 7.12 NestedList Details

```
var nestedlist = new Ext.dataview.NestedList({
    fullscreen : true,
    store      : store,
    title      : 'Authors',
    detailCard : {                                                    // 1
        xtype : 'container',                                          // 2
        tpl   : [                                                     // 3
            'Titter Page: ',
            '<a href="{link}" target="_blank">',
                '{text}',
            '</a>',
            '<br /><br />',
            'Tapping on this link will take you outside of this app'
        ]
    },
    listeners  : {
        leafitemtap : function(nestedlist, list, index, t, record) {  // 4
            var detailCard = nestedlist.getDetailCard();              // 5

            detailCard.setData(record.getData());                     // 6
        }
    }
});
```

{1} Specify the config for detailCard
{2} Use xtype container
{3} Include a template to display data
{4} Add a listener to the leafitemtap
{5} Get the detailCard instance
{6} Apply data to the detailCard

We took the NestedList from Listing 7.12 and added the detailCard configuration **#1**. This is the configuration object that will be used to create the component to be used to display details about the leaf that got tapped on. This detail card will use the xtype `'container'` **#2** and a template using the tpl configuration **#3** to accept data and display it. If we left it here, you would get a blank screen so we need to add a leafitemtap event listener **#4** that will only get fired when you tap on a list item that is a leaf. Within this leafitemtap event listener, we need to get the reference to the detail card using the getDetailCard method **#5** on the NestedList who is passed as an argument on the function. Finally to show data, we execute the setData method on the detail card instance

passing in the data from the record that was tapped on **#6**. Now you should see what is in Figure 7.12.

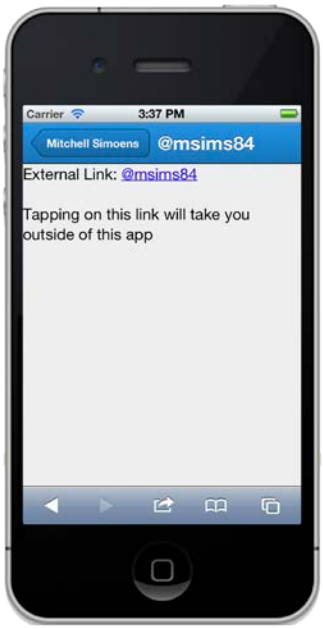

Figure 7.12 View of the detail card in the NestedList

Just like when you tapped on the author it took you to a new `List` and changed the title and the back `Button` in the `TitleBar`, this is the same but instead of a `List`, it used that `detailCard` configuration to create a component. With the logic in the `leafitemtap` event listener, we applied the data from the record we tapped on onto the detail card. Now you have the basis to create a view that can handle hierarchical data and even show a detail view on it. Can you imagine what you can use `NestedList` for? It's powerful but very simple to configure.

7.5 Summary

In this chapter, we added how to use data and what views to use to accomplish your application needs. We first studied how to use the `Store` with its proxy, reader, writer and `Model` classes that it uses. With this knowledge we can use these classes with an understanding of how things work together to be data-driven.

We then learned how a `DataView` worked and what an `XTemplate` is. We implemented some of the more common configurations in order to display data from a `Store` in order to

create something visual. This was very important, as it was our first step into a data-driven view that you will more than likely get to use in your applications.

Next we built upon the `DataView` with the subclass `List` discovering what the differences are. We then implemented an `IndexBar`, grouped our `List` and learned what a disclosure is and how to take action on a user tapping on the disclosure icon. We learned that a `List` can be visually and functionally a very important component to have in our arsenal.

Lastly we learned what hierarchical data looks like and how to consume it with the specialized `TreeStore`. We then used the `NestedList` component to display this data one level at a time and used the `detailCard` configuration in the `NestedList` to display details about a leaf item.

In the next chapter, we will walk through the world of forms with all of the different fields and how to submit data from the form to a remote server.

8 Taking Form

This chapter covers

- Building simple and complex forms
- Loading & Saving of forms
- Binding a form to a list

Over the past chapters we've covered many different ways to manage data and display it on-screen utilizing Sencha Touch's various UI widgets, ranging from Tabs, to Carousels, to XTemplates. While this is an important requirement for pretty much every app out there, I am sure your users are just as demanding as mine, and actually want to interact with the data, maybe even save it! I know, shocking.

As you might have guessed from the title of the chapter, this exact thing is done using form panels and form fields. During the course of this chapter we will take a closer look at the various Form elements, their uses, and how to put them together in a form that can be loaded and saved. After that we will explore ways to use forms in conjunction with other components, like the List sample we just finished in chapter 5. So, what are you waiting for? Let's get started.

8.1 What makes Form Panels so special anyway?

It should come as no surprise that developing and designing forms represents one of the commons tasks in every app developer's life, and as such, we as developers are a demanding bunch. A way to load the form? Yes sir. A way to save it? Of course. Validation? An absolute must!

With the bar set high, it is only natural for Sencha to tap into the extensive set of Widgets developed for Ext JS, Sencha Touch's bigger cousin, and bring some of those

features over to provide us with an impressive list of different form elements. Figure 8.1 gives us a simple overview of all the Forms related UI Widgets, and their inheritance chain.

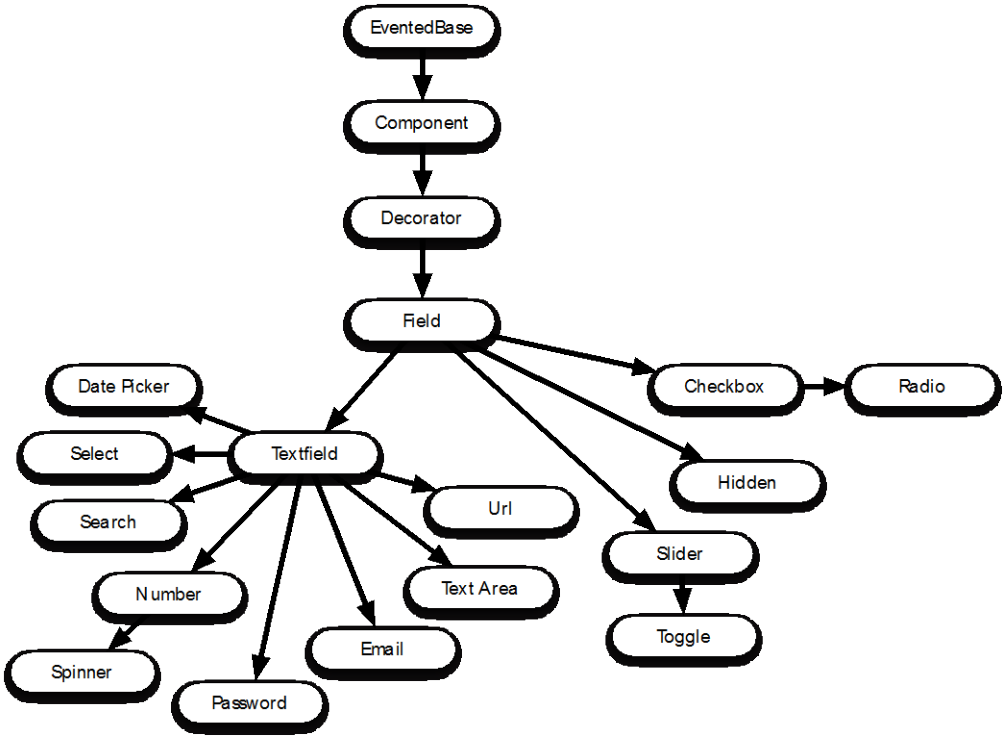

Figure 8.1 Forms Related Widgets and their inheritance.

At its simplest, forms in Sencha Touch are comprised of a container, the `FormPanel` class, that gets filled with one of the various UI widgets ranging from text fields, to date pickers, to sliders, all of which extend the `Field` class as depicted in Figure 8.1 above.

The `FormPanel` follows our by now familiar Component Model pattern and extends the base `Panel` class as shown in Figure 8.2 below. As such, it has all the standard capabilities of a `Panel`, like layouts, ways to manage children, and a few additional methods to handle submitting and loading of form data.

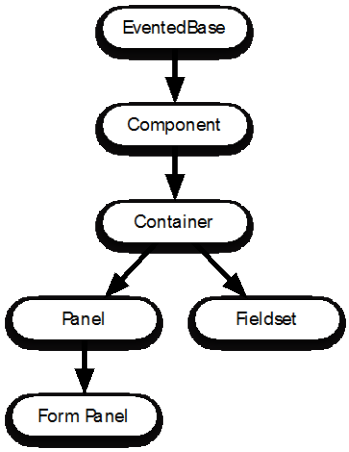

Figure 8.2 Inheritance chain for Form Panels and Fieldset.

The `Field` class in turn extends the `Decorator` class, which simply provides extra visual decoration around the base `Component` class and provides functionality to render labels next to the form fields, get and set the value of a form field, or even mask the field during load operations.

Back in chapter 3, we talked about the Component Model and the Stability and predictability it introduces into the framework. The relationship between the `FormPanel` and the `Field` class is a prime example to highlight this paradigm. The Field class exposes several functions to allow getting, setting, or resetting the value of the field. In addition, it tracks whether the field is "dirty", which refers to whether the field value has changed since the initial load. The `FormPanel` understands these things about every field intrinsically, and knows how to utilize these methods to automatically populate any of its child fields whenever the form is loaded or submitted for example. This is possible because each field instance exposes these same functions, creating the before mentioned predictability.

"Sencha Touch" !== "Ext JS"

For those developers coming over from the Ext JS side of things, you will find that forms in Sencha Touch deviate slightly from their Ext JS variant. These deviations range from a drastically different, dare I say prettier, visual design to a more simplistic layout that better accommodates touch input. Under the hood, there are various changes as well. As of this writing, there is no built-in validation for forms, and the load method that was your bread and butter for populating forms with data has been replaced with a methodology that forces you to use models instead.

Now that we have a basic overview of Forms and their relationship to Fields, it is time we actually put this knowledge into practice and build a form.

8.2 Building a Simple Form

Like all the other `Container` subclasses, the `FormPanel` class can leverage any layout that's available in the framework to create exquisitely laid-out forms. Before we go crazy with multi-column, or multi-tabbed forms, something that requires us to go through quite a few design and usability considerations, we'll stick to the basics and use the default `VBoxLayout` to stack form elements one on top of another.

In Listing 8.1 below we will put together a simple form that could be used as a hypothetical counter piece to edit the `ListView` data from our code in Chapter 6, listing 6.x.

Listing 8.1 – A simple form

```
var myForm = Ext.create("Ext.form.Panel", {                          1
   fullscreen: true,
   items: [
       {
            xtype: "textfield",                                      2
            label: "First",
            name: "firstName",
            placeHolder: "Enter First Name Here"                     3
       }, {
            xtype: "textfield",
            label: "Last",
            name: "lastName",
            placeHolder: "Enter Last Name Here"
       }, {
            xtype: "selectfield",
            label: "Status",
            name: "inviteStatus",
            placeholder: "nothing",
            options: [                                               4
                { text: 'Undecided', value: 'undecided' },
                { text: 'Accepted', value: 'accept' },
                { text: 'Declined', value: 'decline' }
            ]
       }
   ]
});
```

1. **Instantiating a form Panel**
2. **Instantiating a Text Field**
3. **Placeholder text when a field is empty**
4. **Defining Options for a Select Field**

The code itself should be pretty straightforward by now. We first start by creating a `FormPanel` instance **(1)** and giving it the `fullscreen` attribute. This works since the `FormPanel` is simply an extension of the `Panel` class, and as such the `fullscreen` option will make it occupy the entire screen. Note that without this, the form would not show

on the screen, since Sencha Touch always requires one container to take up the entire screen by using the fullscreen option. For the child elements of the form we define two `textfields` **(2)**, one for first name and one for last name. Notice that we give each `textfield` a `placeholder` text **(3)** that will automatically be shown whenever the text field is empty. In general, it is best to try and make the `placeholder` text as useful and instructive as possible to give users an idea of what they are supposed to do within your form. Lastly we define a `SelectField` and give it three `options` **(4)** that, depending on the platform you're running this on, will show up either as an action sheet or a dropdown when the field is activated. Once we execute the code from Listing 8.1 we should get a result that looks like Figure 8.3 here:

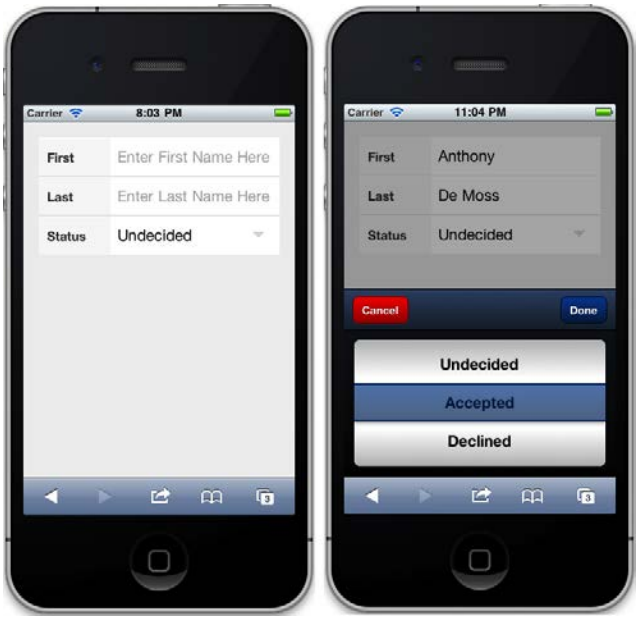

Figure 8.3 Simple Form Sample on the left, and the same form with text fields filled in and select field activated on the right.

In case you are wondering about the one property we didn't talk about yet, the `name` property, it serves one simple purpose: to provide an identifier for each field, which the `FormPanel` uses whenever we load data into the form, or retrieve data from the form.

During load operations, a JSON object containing name/value pairs is used. The name of each property in the JSON object is mapped to a field that shares the same name within the `FormPanel`. The form automatically handles assigning the values from the JSON object to

each respective field. A sample JSON object we could use to load data into the `form` from listing 8.1 looks like the following:

```
{
    firstName: "Anthony",
    lastName: "De Moss",
    inviteStatus: "accept"
}
```

On the flip side, retrieving values works almost in the exact reverse. The form exposes a convenient `getValues` method that returns a JavaScript object containing name/value pairs for each field in the form. Running the following line in the Safari Debug Console would return a JavaScript object that looks exactly like the one above we used for loading.

```
myForm.getValues();
```

I assume you're sitting there now, wondering just exactly how loading & submitting of form data works. Don't fret; we will cover this topic in greater detail in later parts of this chapter after we've gotten a better understanding of all the different form elements and the caveats to keep in mind about each.

8.3 An overview of the different Form Widgets

Sencha Touch offers quite a lot of wrapped native HTML5 input fields, as well as a few custom widgets. The Textfield, Checkbox, URL Field, Email, Textarea, Number Field, Password and Radio fields all implement native HTML5 input elements simply with additional styling to make everything look the "Sencha" way. Each one of these widgets, with the exception of the Radio and Checkbox fields, will force the native slide in keyboard to appear when focused, thus allowing users to enter data into the field. Figure 8.4 (below), illustrates what most of these input fields look like when rendered on the screen.

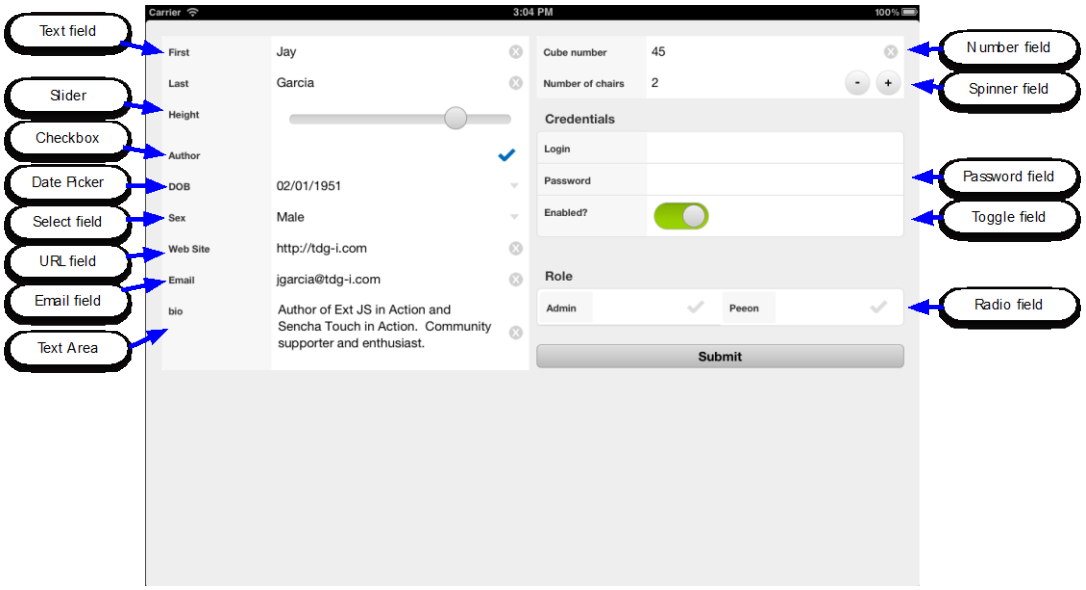

Figure 8.4 Overview of the different Form Widgets

Now that we're equipped with some visuals, let's explore each one of these form elements in more detail and get a better understanding of how to instantiate it, and note any caveats you should be aware of.

8.3.1 Text Field

The Text Field provides the base for quite a few of the other fields on this list. It simply wraps the standard HTML5 input field and exposes a set of convenient configuration options to customize the behavior to some degree or another. Instantiation is pretty straightforward and looks like the following:

```
{
     xtype: "textfield",
     label: "Text Field",
     placeHolder: "Something goes here",
     value: "This is a text field",
     required: true,
     autoCorrect: true,
     autoCapitalize: true,
     autoComplete: true,
     maxLength: 5
}
```

The `label` provides a convenient way to mark what your field is all about, and is automatically attached to the left side of your field. Each and every form element can have a label associated with it. Keep in mind that the device you're using limits the size of a label. On a phone for example, your labels are significantly smaller than on a tablet device, causing long text in the label to automatically wrap, which can look quite ugly.

All the text-based fields (URL field, email field, text area, select field) provide for a `placeholder` text that is automatically shown whenever the field is empty. The `placeholder` can be used to augment the label and provide additional instructions to the user. Generally, it is not advisable to forgo the label in lieu of just a placeholder, since the placeholder disappears once a field is filled out, creating a scenario where the user has to guess the purpose of the field.

The `required` parameter will be one of the more confusing ones, especially for those developers coming from Ext JS. As of this writing, Sencha Touch does not provide any built-in form validation, so this parameter simply marks a field as required by adding an asterisk into the label. There is no additional visual feedback if a field is left blank, or to prevent a form from being submitted.

The `maxLength` parameter allows you to limit the amount of characters that can be entered into a text field. Once the limit has been reached, any additional characters typed on the keyboard are simply discarded. If you manually set the value of a `textfield` through the `value` config parameter or the `setValue` function, the `maxLength` parameter is circumvented and it is possible to add an arbitrary amount of characters to the field.

The last three configuration options deal with auto correction, auto capitalization, and auto complete. The device you're using determines the usefulness of these options. On an iOS device (iPhone / iPad) auto capitalization will automatically enable the shift key for the first character, and auto correction will trigger the annoying correction bubble that shows up to suggest "better" spelling for your word.

8.3.2 URL Field

The URL Field extends the basic text field and brings with it all the same configuration options. Under the hood, the HTML5 input field is set to type "url". The only real purpose this serves is that, depending on your device, the keyboard will automatically be put into "URL" mode, showing keys that are more useful for entering a URL, when editing the field. From an implementation perspective, the code looks like the following:

```
{
     xtype: "urlfield",
     label: "Url",
     value: "http://www.senchatouchinaction.com"
}
```

Since the `urlfield` is based on the `textfield`, you could have used any of the other configuration options we used for the text field. It is important to note is that this field does not provide any validation to ensure that the URL is actually properly formatted.

8.3.3 Email Field

The Email Field acts as a close cousin to the URL field and follows the same principle, except for emails. Whenever the field is edited, the on-screen keyboard is switched to "email" mode to make it easier to enter an email address. As with the URL field, you could use any other configuration option from the text field. Follow this code to instantiate an email field:

```
{
     xtype: "emailfield",
     label: "Email",
     value: "ademoss@topshelfsolution.com"
}
```

Just like the URL field, the email field does not provide any validation to ensure that the email address is indeed valid, or even formatted correctly.

8.3.4 Number Field

The Number Field is yet another field similar to the Email and URL Field. It extends `TextField` and provides additional configuration options for a minimum and maximum value, as well as a step value. At this point, these options are really only relevant if you plan on using your app in a non-mobile browser. Mobile browsers will render this field as a plain text field that brings up the numeric keyboard when the field is edited. Instantiation follows almost the same code as the text field before and looks like this:

```
{
     xtype: "numberfield",
     label: "Number",
     minValue: 1,
     maxValue: 10,
     stepValue: 1,
     value: 7
}
```

Since as of this writing, HTML5 is still not properly implemented in most browsers, especially as far as field validations are concerned, it would normally be possible to enter any arbitrary value into the field, instead of being restricted to only numbers. Fortunately for us however, Sencha Touch comes to the rescue and does basic validation to ensure the entered value is indeed a number.

8.3.5 Password Field

The password field is where we start seeing our first difference from the text field. The field automatically obscures any user-entered text as asterisk characters. The same is true for manually assigned values either through the `value` config option or the `setValue` function. Both of those will be obfuscated as well.

```
{
     xtype: "passwordfield",
     label: "Password",
     value: "abc"
}
```

One of the more useful options for the password field is the fact that it allows for the same configuration options as the text field, including the placeholder option, which will show up as plain text instead of starred out characters.

8.3.6 Text Area

The text area is very similar to the text field, the main difference being that the text area allows for multi-line text, which is something the text field does not. Newline characters ("\n") are automatically interpreted by the field and will insert a new line into the text. Implementation follows the same principle as the text field and looks just as easy:

```
{
     xtype: "textareafield",
     label: "Text Area",
     maxRows: 5,
     value: "This is a larger text area.\n\nWe can even get multiple lines
in here"
}
```

Notice the maxRows config option: it can be used to limit the amount of rows you wish the user to be able to enter.

8.3.7 Checkbox Field

The Checkbox field works similarly to its native-web counterpart, except it is stylized via Sencha Touch's own check icon to mimic native application behavior. Instantiation is simple, and only consists of a `label` for the most part. It is possible to set the initial value of the checkbox via the `checked` option.

```
{
     xtype: "checkbox",
     label: "checkbox - ready",
     value: "yes",
     checked: true
}
```

The `value` option provides you with a way to set the value that will be submitted if the checkbox is checked and a form submit occurs. In the above example, the value "yes" would be submitted, instead of the default "true" value.

8.3.8 Radio Field

The Radio field follows in the footsteps of the checkbox field, both visually and under-the-hood. It extends checkbox and in fact looks the same once rendered. The only difference to the checkbox is the behavior when a radio field is activated, as it switches between the different grouped fields, only allowing one of them to be checked at a time. To see an example of this, take a look at the following code:

```
{
     xtype: "radio",
     name: "myradio",
     label: "radio - one"
}, {
     xtype: "radio",
     name: "myradio",
     label: "radio - two"
}
```

This provides us with two radio fields that are automatically grouped together (not visually, but from a selection perspective). The grouping happens based on the name of the field. Any radio field that shares the same name with an already existing radio field is automatically added to the group. Whenever you click one of the two fields, the other one becomes unchecked.

8.3.9 DatePicker Field

The DatePicker field gives your users the ability to choose a date from a set by mimicking the native iOS Date Picker input widget. The `DatePicker` field implements a `Sheet`, which is an overlay that slides in from the bottom, allowing the user to select values via vertical swipe or "flick" gestures. No matter what the device or its orientation, the `DatePicker` field will always display a sheet, forcing selection though this modal overlay. The following code illustrates how to instantiate a `DatePicker` field.

```
{
     xtype: "datepickerfield",
     label: "Pick a Date",
     value: { year: 2011, month: 2, day: 23 }
}
```

It is possible to set the initial value of the field via the `value` config option or the `setValue` method. Both ways accept either a JavaScript date object in the form of `new Date()` or a custom JavaScript object containing a year, month, and day property like the sample code above.

8.3.10 Spinner Field

The Spinner field is a custom styled input field, allowing users to enter numeric values, much like the Number field, with the addition of easy-to-use decrement and increment buttons on the side of the field. Notice that the increment and decrement buttons are larger than standard html spinner buttons, making them easier to use on a touch device. Implementation follows largely the same parameters as the number field:

```
{
     xtype: "spinnerfield",
     label: "Spinner",
     minValue: 1,
     maxValue: 10,
     increment: 2,
     cycle: true,
     value: 9
}
```

The main difference in config options, compared to the number field, is the addition of a `cycle` property. This causes the value to "wrap" around if the min or max is reached. For example, assume the current value is 9, the max is 10, and we have an increment of 2. The next click to increase the value would bring it over 10, thus causing it to wrap around to the bottom and start over at 1. Unlike the number field, the spinner field does not allow the user to enter values directly, instead forcing a value change through the increment and decrement buttons.

8.3.11 Slider Field

The Slider field implements native Sencha Touch Draggable and Droppable classes, allowing users to input a numeric value, via a swipe and tap gestures. The slider can be restricted via the `minValue` and `maxValue` options, as well as the `increment` option. In code that looks like the following:

```
{
     xtype: "sliderfield",
     label: "Slide me",
     minValue: 20,
     maxValue: 100,
     value: 50,
     increment: 5
}
```

The `minValue` and `maxValue` do not change how far the slider can move, but only change what the left edge and right edge represent respectively. The `increment` option changes where the slider snaps to when moved. This takes into account where you drop the slider, and automatically rounds to the nearest number that represents the next increment. One of the major drawbacks of the slider is the lack of visual feedback about its current value.

8.3.12 Toggle Field

The Toggle field extends `Slider`, allowing users to toggle the field like an on/off switch via a swipe and tap gesture. Under the hood the toggle field is just a slider with a `minValue` of 0 and a `maxValue` of 1.

```
{
     xtype: "togglefield",
     label: "Toggle Me",
     value: 1
}
```

You can set the initial state of the `ToggleField` to on or off by setting the `value` field to 1 or 0 respectively.

8.3.13 Select Field

The Select field is the closest thing to a traditional Combo Box / Dropdown you will find in the Sencha Touch arsenal. It allows you to present the user with a list of choices to pick from, and whichever the user selects will be shown in the field. Depending on the device, the `SelectField` displays different input widgets. On a phone, it uses a `Sheet` that slides in from the bottom, similarly to the `DatePicker`, and on a tablet it uses a small dialog-type control containing the options to select from. Here is an example implementation of a `SelectField` using hard-coded options:

```
{
     xtype: "selectfield",
     label: "Select",
     options: [
         {
             text: 'Yes',
             value: 'yes'
         }, {
             text: 'No',
             value: 'no'
         }
     ]
}
```

In this case, the defined options never change. Each option must have a `text` and a `value` field, where the `text` property represents the text that is shown in the list, and the `value` property represents the value that will be submitted when the form is submitted.

Quite frequently, you will find yourself in a situation where your data is not formatted to contain a `text` or `value` field, or where hard-coding the options is simply not plausible. For those instances, the `SelectField` provides additional options in the form of a `displayField` and a `valueField`. These are strings that determine which property from your data should be used instead of the `text` and `value` property.

To fill your Select Field with dynamic data, you can employ a `DataStore`, which triggers a remote load. In practice, it looks like the code for listing 8.2.

Listing 8.2 Dynamically Loaded Select Field

```
Ext.define('Person', {
     extend: 'Ext.data.Model',
     config: {
         fields : [                                                    1
             { name: 'name' },
             { name: 'pID' }
         ]
     }
});


var peopleStore = Ext.create('Ext.data.Store', {
     model : 'Person',                                                 2
     autoLoad : true,
     proxy: {
         type  : 'ajax',
         url: 'getPersons.json',                                       3
         reader: {
             type: 'json',
             rootProperty: 'data'
         }
     }
});


var myPanel = Ext.create("Ext.form.FormPanel", {
     fullscreen  : true,
     items : [
         {
             xtype : 'toolbar',
             title : 'Tell us about yourself.',
             docked : 'top'
         },
         {
             xtype: 'selectfield',
             label: 'Select Name',
             valueField: 'pID',                                        4
             displayField: 'name',
             store: peopleStore
         }
     ]
});
```

1. **Defining a model with two fields**
2. **Using our model for the Store**
3. **Setting the url of the Store**
4. **Setting the valueField and displayField**

A lot happens in listing 8.2 simply to load data into the select field from a remote source. We start by defining a `Model` **(1)** with two fields. In this case we are representing people, so we give everyone a `name`, and a `pID` (short for personal id) that will serve as a unique identifier for each record. Of course, we need to put our model to good use, so we continue with a `Store` that points to our model **(2)**. The store alone won't know how to get to the data, because of that we need to define a `Proxy` that tells the store from where and how the data is going to be loaded. The where is handled through the `url` property **(3)** which points to a static JSON file. The how is handled through the `reader`, defined on the `Proxy`, that uses a `type` of `json` which just so happens to be the format of our data. Use the following dummy data and save it in a file named "getPersons.json".

```
{
     success: true,
     data: [
         { name: "Anthony De Moss", pID: 1 },
         { name: "Jay Garcia", pID: 2 }
     ]
}
```

After we're done setting up the model and store, we move on to create a form panel with a `SelectField` and point it at the store we just created. We already know that our data won't have the standard `text/value` properties that the `SelectField` expects, so we employ the `valueField` **(4)** and the `displayField` and point them at the `pID` and `name` respectively. This way the `name` is shown in the actual field, and the `pID` would be used if we were to submit the form. That's all there is to loading a store dynamically.

By now you should have a good overview of the different form elements and how to go about creating a simple form as well as dynamically loading content into a select field. Obviously, nothing is ever simple in life, and the likelihood of being able to accomplish everything you need with a plain and simple form is virtually non-existent. This is where some of the advanced techniques like multi-column forms and field sets from the next section will come into play.

8.4 Building Complex Forms

Most of the forms we have built so far all have one thing in common; they follow the most basic design possible, stacking form elements in a single column, one on top of the other. This paradigm has certain advantages, especially when considering the fact that your form most likely has to work on multiple different devices, such as a phone or a tablet. For that reason, a single column is the perfect fit since it simply works across the spectrum of devices. This doesn't mean however, that your `Form` has to be one long and boring list of stacked elements. Sencha Touch provides for an easy way to split up your form visually, by grouping elements into separate Child-`Containers` with a `title` and `instructions`, via the use of a `Fieldset`.

8.4.1 More organized Forms with Fieldsets

`Fieldsets` extend the `Container` class, and as such you can think of them simply as `Containers` with a title. In practice, the result of using a `Fieldset` looks like the right side of figure 8.5 below versus the standard stacked form on the left.

Figure 8.5 Form without Fieldsets on the left. Same Form with Fieldsets and instructions on the right.

The way the `Form` is broken up into smaller pieces should make the benefits of using a `Fieldset` immediately apparent. Smaller pieces are more easily digestible by the end-user, and thus provide improved usability and flow. Simply place the following code within the `items` array of a `FormPanel` to see it in action for yourself.

```
{
     xtype  : 'fieldset',
     title  : 'Personal Info',
     instructions: "Enter your personal information",
     items  : [
         {
             xtype: 'textfield',
```

```
                label: 'First',
                name:'firstname',
                placeHolder: "Enter your first name"
            },{
                xtype: 'textfield',
                label: 'Last',
                name:'lastname',
                placeHolder: "Enter your last name"
            }
        ]
    }
```

The code follows most of the samples we've already covered. We start by defining the `Fieldset` using an XType, and give it a `title` and `instructions`. Based on figure 8.5 you can see that the `title` shows up above the first field, and the `instructions` are shown below the last field. Although figure 8.5 doesn't illustrate this perfectly, `instructions` are centered and automatically wrapped to additional lines if necessary. Since the `Fieldset` is itself a `Container`, it has its own `items` array where you place the form elements you wish to include. In this case we have two `textfields` with a `label` and `placeholder` text.

8.4.2 Multi-column Forms

While we've established that using `Fieldsets` can dramatically improve visuals of your forms, we still have one major problem: any tablet device wastes a sizeable amount of screen real estate on very wide fields when all the fields are organized in a single column. For such purposes, it is handy to understand that `Forms` (and `Fieldsets`) are simply `Containers` that can contain their own `Layout`. Knowing this, we can create multi-column forms that are more specifically tailored towards tablet devices with larger screens.

Mobile phones are cramped with tiny screens

It is important to remember that no matter how big the screen is on a mobile phone, it is always going to be cramped and have less screen real estate than a tablet device. Due to this, multi-column layouts are rarely suitable for phone devices, something that should be evident when taking a closer look at how cramped the labels are in Figure 8.5.

The next sample ventures into the world of multi-column forms. Take a sneak peak at figure 8.6 to see what we're building. Notice that this sample is made specifically for tablet devices and will not work properly on mobile phones due to space constraints.

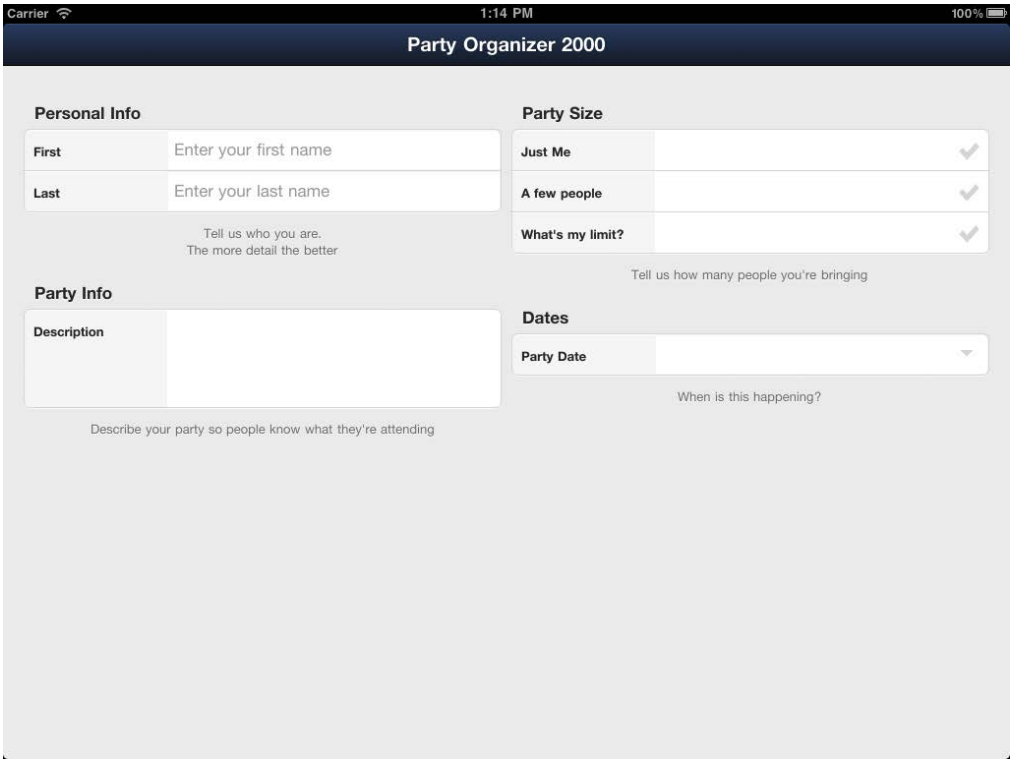

Figure 8.6 A sneak peak at the Multi-Column form with Fieldsets we're going to build

Although all four of the `FieldSets` needed aren't terribly complex, we will nonetheless start out by first defining each one separately before finally assembling it all together into a full `FormPanel` at the end. The first two `FieldSets` (listing 8.3) represent the left column, which consists of two `textfields` and a `textarea`.

Listing 8.3 Creating the Fieldsets for the Left Column

```
var fieldset1 = {
    xtype  : 'fieldset',                                                    1
    title  : 'Personal Info',                                               2
    instructions: "Tell us who you are.<br />The more detail the better", 2
    items  : [
        {
            xtype: 'textfield',                                             3
            label: 'First',
            name: 'firstname',
            placeHolder: "Enter your first name"
        },{
            xtype: 'textfield',                                             3
```

```
            label: 'Last',
            name:' lastname',
            placeHolder: "Enter your last name"
        }
    ]
};

var fieldset2 = {                                                    4
    xtype  : 'fieldset',
    title  : 'Party Info',
    instructions: "Describe your party so people know what they're
attending",
    items  : [
        {
            xtype: "textarea",
            label: "Description",
            name: "description"
        }
    ]
};
```

1. **Set the XType to fieldset**
2. **Provide a title and instructions**
3. **Define the textfields needed**
4. **Define the second fieldset**

For the most part, we keep it short and simple by defining both `Fieldsets` via XType configuration **(1)**, and providing a `title` **(2)** and `instructions` **(2)** on each `FieldSet`. Notice that the `instructions` on the first `FieldSet` contain an html line break to force the instructions into two lines. You could easily use any other html markup here to mark text in bold, italics, underlined or pepper it CSS classes. We then proceed to populate the first `FieldSet` with two `textfields` **(3)**, each of which receives a `label`, as well as `placeholder` text to provide even more ways to tell the user what to do. The second `FieldSet` **(4)** follows the same basic steps, and simply contains a barebones `textarea`.

From here we'll continue into both `FieldSets` in the right column, which you can see in listing 8.4.

Listing 8.4 Creating the Second Fieldset

```
var fieldset3 = {
    xtype  : 'fieldset',                                             1
    title  : 'Party Size',
    instructions: "Tell us how many people you're bringing",
    items  : [
        {
            xtype: 'radio',
            name: 'size',
            label: 'Just Me',
            value: 'small'
        },{
            xtype: 'radio',
            name: 'size',
```

```
            label: 'A few people',                                    2
            value: 'medium'
        },{
            xtype: 'radio',
            name: 'size',
            label: "What's my limit?",
            value: 'large'
        }
    ]
};

var fieldset4 = {                                                     3
    xtype  : 'fieldset',
    title  : 'Dates',
    instructions: "When is this happening?",
    items  : [
        {
            xtype: 'datepickerfield',
            label: 'Party Date',
            name: 'partydate'
        }
    ]
};
```

1. **Fieldset with three radio fields**
2. **Give each radio field the same name**
3. **Fieldset with a Date Picker**

As before, listing 8.4 starts out with the initial `FieldSet` **(1)**, which we populate with three `Radio` fields. Notice that each one of the `Radio` fields has the same `name` **(2)** in order to mark them as a set. Last but not least, we define the final `FieldSet` **(3)** and give it a `DatePicker` field that starts out blank.

This marks the last step in setting up all the `Components` required to generate both columns. We're now ready to put it all together (listing 8.5) and construct the actual `FormPanel` itself that will house everything.

Listing 8.5 Putting it all together

```
var myPanel = Ext.create("Ext.form.Panel", {                          1
    fullscreen  : true,
    scrollable  : "vertical",
    layout      : {                                                   2
        type  : 'hbox',
        align : 'stretch'
    },
    defaults    : {                                                   3
        flex : 1,
        style : 'padding: 5px;'
    },
    items : [
        {
            xtype : 'toolbar',
```

```
            title : 'Party Organizer 2000',
            docked : 'top'
        },
        {
            xtype  : 'container',                                    4
            items  : [
                   fieldset1,                                        5
                   fieldset2
            ]
        },{
            xtype  : 'container',
            items  : [
                   fieldset3,
                   fieldset4
            ]
        }
    ]
});
```

1. **Start with the Form Panel**
2. **Use HBOX layout for the columns**
3. **Set default values for each column**
4. **Define a container for each column**
5. **Place the fieldsets in the container**

Here we are finally getting to creating our `FormPanel` **(1)** and putting everything together. Since the `FormPanel` is nothing more than a special `Container`, we start by deviating from the standard `VBoxLayout`, and give the `FormPanel` an `HBoxLayout` **(2)** with an `align:stretch` option instead. If you think back to chapter 4, the `HBoxLayout` provides us with a way to have multiple columns in a `Container`, and the `align` option is used for the "vertical alignment", which in the case of `stretch` overrides the height, forcing each column to occupy the full height of the `FormPanel`. We uniformly size each column horizontally through the use the `flex` property, which we stick into a `defaults` object **(3)** so it gets automatically applied to each column. In addition, we don't want the two columns to touch each other, so we set an additional padding of 5 pixels via the `style` property.

Next, we move on to defining the actual `Containers` **(4)**, which are downright simplistic compared to everything else. An `xtype` definition, as well as an `items` array that contains the `FieldSets` **(5)** we want in each column does the trick. Each `Container` will automatically use the default `VBoxLayout` and stack the `FieldSets` one on top of the other. In the end, we come out with a form that looks like Figure 8.6.

8.4.3 Doing more with our Multi-column Form

Before we move on to loading content and managing data within forms, let's chat about some of the possibilities we didn't cover. The form we just build is multi-column through the use of the `HBoxLayout` and `Containers`. Instead of doing that, we could have just as well designed the form differently and used `FieldSets` that span the entire width of the form, but contain two columns within each `FieldSet`. Since a `FieldSet` extends the

Container class, it can take any of the Layouts we covered in Chapter 4, and as such we could define an HBoxLayout within a FieldSet and then populate it with two Containers, each of which contains the form Components we want in each column. The possible combinations we could achieve are virtually endless. The one caveat you always have to keep in mind however is that you should never nest FormPanels within each other. There is no necessity or benefit to it, and in fact it causes tremendous problems that are hard to debug. If you do require multiple FormPanels on a screen, simply make sure to keep them separate.

By now you've seen how combining multiple Components, Field Sets and layouts can result in something that's both usable and space saving. With that out of the way, it is time we actually inject some usefulness into form and talk about ways to load and submit data using forms.

8.5 Managing Data with Models

Submitting and loading data is one of the most crucial parts about forms and incidentally one of the areas most new developers commonly get tripped up on. Sencha Touch form submission requires a bit of rethinking from the old school, page-refresh-type form submission a lot of developers are used to. For those developers coming from the Ext JS side, form submission will be a walk in the park. On the flip side, loading a form has changed quite dramatically, and will most likely require getting used to by just about anyone.

8.5.1 Submitting Data

Submitting a form requires only a few steps you need to be aware of. The first is that you need to get a handle on your FormPanel. Simply use one of the component query methods we discussed in chapter 5, or assign the form to a variable the way we did in listing 8.5. The second thing you we need to understand is the way the submit method works. For that, take a look at the following listing, which is based on the code from listing 8.5.

Listing 8.6 Submitting our form

```
var formSubmit = function() {
     myForm.submit({
            url: "success.json",                                    1
            success: function(form, response) {                     2
                   Ext.Msg.alert("Success", response.msg);
            },
            failure: function(form, response) {                     2
                   Ext.Msg.alert("Failure", response.msg);
            }
     });
}
```

1. **Call submit function with URL to submit to**
2. **Handle a successful & failured submit**

In listing 8.6 we create a `formSubmit` function that handles submission of your form and could be called from anywhere else in the code. Our handler function uses the `form.submit` method to take care of actually sending the data. The format is pretty much the same as it is in Ext JS, and consists of a `url` **(1)** that determines where the form should be submitted to, as well as a `success` **(2)** and `failure` handler.

Although we specify the URL in the submit call, you should be aware that we could have just as easily used the `url` configuration option on the form panel, which is automatically used when the `submit` method is called. The only reason we didn't do that in this sample is to illustrate the fact that the URL can be changed at runtime.

The `success` and `failure` handlers are callback functions, called if the form submission was successful or failed respectively. At a bare minimum, your backend needs to return a JSON response containing a boolean `success` property, which influences which callback is triggered. Our particular example expects an additional `msg` field to let the user know what is going on:

```
{success: true, msg: "Thank you for your submission."}
```

Likewise, if you server-side code deems the submission as unsuccessful for any reason, the server should return a JSON object with success property set to false. Unlike ExtJS, there is no way in Sencha Touch to provide additional feedback or validation information for individual form fields. You will simply have to roll your own in this case and handle that in your callback.

Handling Form Validation

A common paradigm is to validate the data in your form before submitting. This ensures that all required fields are filled out, and any specific validations like an email field actually containing an email address are true. In Ext JS, this was a breeze using the `form.validate` function. As of this writing, Sencha Touch does not have any equivalent function in its arsenal, forcing you to get creative. One way around this is to put validations on the model, like we covered in chapter 6, and then before submitting, pulling the data out of the form using the `getValues` function and feeding it into a new instance of the model. This allows you to leverage the model validators to validate your form.

Alternatively, you can visit the website for this book at www.senchatouchinaction.com where you can find additional plugins and extensions that alleviate this problem as well.

8.5.2 Loading Data into your form

The use case for just about every form includes saving and loading data. In the good ol' Ext JS days, that was simply done using the load method, which follows the same paradigm as submitting data with the `submit` method. Sencha Touch however is not like Ext JS in this

regard, removing the generic `load` method, in lieu of a `setRecord` method that consumes a model instance. This new methodology forces you to use a `model` and `proxy` to load data first, and then feed it into the form. To illustrate this concept, we will once again revisit code from listing 8.5. Before we can begin however, we must have data to load, so we'll dive right into creating some sample data to load. Simply save the following JSON data in a file called data.txt.

```
[{
     firstname: "Anthony",
     lastname: "De Moss",
     description: "Sencha Touch In Action Launch Party!!",
     size: "medium",
     partydate: {
            year: 2011,
            month: 2,
            day: 23
     }
}]
```

Unlike the response for the form submission, the `model.load` method does not require a `success` property to be present in the JSON response, but instead expects the data to be wrapped in additional square brackets to create an array of records. In order to consume the JSON object above, we need to create a model first.

```
var personModel = Ext.define("Person", {
     extend: 'Ext.data.Model',
     config: {
         fields : [
             "firstname", "lastname", "description", "size", "partydate"
         ],
         proxy: {
             type: 'ajax',
             url : 'data.txt ',
             reader: 'json'
         }
     }
});
```

We register our model and give it a name, and fields that correspond to the names of the form elements from listing 8.5. We define an Ajax `proxy` on the model itself, and point it to the data.txt file that we created earlier. Notice that we assign the return value of `Ext.regModel` to a variable we name `personModel`. This gives us a few benefits when doing the load. Each registered model exposes a few functions that can be called without needing an instance of the model, requiring however that you have a handle on the registered model itself. Alternatively, we could have used `Ext.ModelManager.getModel` instead and fed it the name of our model to get a handle on the model itself. Next up is the function that actually handles the loading.

```
function loadHandler() {
     personModel.load(123, {
            success: function(theRecord) {
                   myForm.setRecord(theRecord);
            }
     });
}
```

Here we create a `loadHandler` function to take care of loading data remotely via the `model` we registered. We try to keep this function somewhat generic so it could easily be called from anywhere in the code. We reference our `personModel` we stored earlier, and call the `load` method. The `load` method expects two parameters, the first being the unique id of the record you want to load, and the second being a configuration object that contains a success handler and a failure handler, to be triggered based on whether the load operation completed successfully or not.

The `success` handler function receives the loaded record (an instance of our model) as its only parameter. We in turn feed that into the `form` using the `setRecord` method. Et voila! We have data in our form, just like figure 8.7 below.

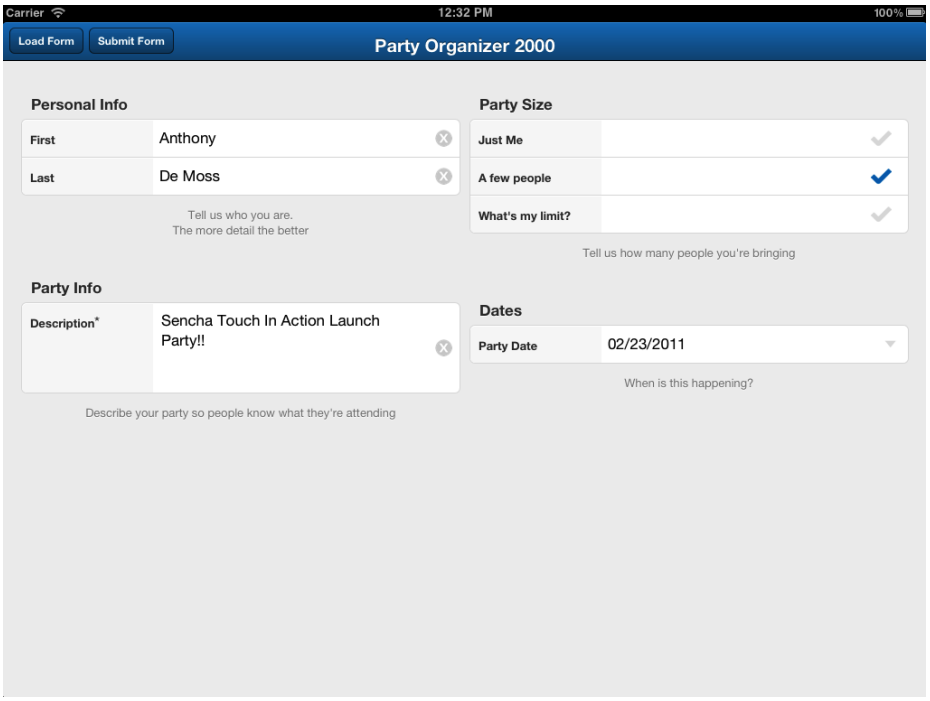

Figure 8.7 Our multi-column form, loaded with data

In the event you already have the data on hand, let's say because you made a separate Ajax call, or from another component such as a list, you can set the values of the form fields using `myForm.setValues(dataObj)`. Where `dataObj` represents a JSON object equivalent to the content we stored in data.txt, minus the square brackets.

> TIP: To retrieve the values from any given form, call `getValues` from the `FormPanel` instance. For example `myForm.getValues()` would return an object containing keys representing the names of the fields and their values.

Loading data can be as simple as that.

Congratulations! You've now configured your first truly complex form, learned how to load and save its data, and pretty much covered all the basics of forms in general. It is time we put all this knowledge together, and see what we can do when we use forms with other components, like a list.

8.6 Binding a Form to a List

Standalone samples and isolated test cases are a great way to illustrate a particular concept, but let's be honest; they never really cover the interesting stuff, or represent more real-world type scenarios. Our next sample should remedy that fact to some degree or another by utilizing portions of the list sample from chapter 6, panels from chapter 4, as well as code from listing 8.1 in this chapter. The app we're building is a simple Party Invitation manager (figure 8.8) that shows a list of invitees, which are bound to a form to allow editing.

Figure 8.8 The different screens for the Party Invitation Manager.

While this scenario is still contrived, it represents a starting point that could easily be adapted for other situations. Figure 8.8 illustrates the different screens our app will have, beginning with the list view on the left, to the edit screen in the middle, to the exposed picker for the invitation status on the right. Hitting save within the form posts the changes back to the list and makes them visible in real time. To build this app, we will start out with the data portion first.

Listing 8.7 Setting up Data Handling for the Party List

```
Ext.define("InviteList", {                                          1
     extend : "Ext.data.Model",
     config: {
         fields : ['firstName', 'lastName', 'inviteStatus']
     }
});

var inviteData = [                                                  2
    {
            firstName : 'Jay',
            lastName  : 'Garcia',
            inviteStatus  : 'decline'
     }, {
            firstName : 'Anthony',
```

```
            lastName  : 'DeMoss',
            inviteStatus  : 'undecided'
    }, {
            firstName : 'Sebastian',
            lastName  : 'Stirling',
            inviteStatus  : 'accept'
    }
]

var theListStore = Ext.create('Ext.data.Store', {                    3
    model : 'InviteList',
    data : inviteData,
    autoLoad : true
});
```

1. **Define the model for the List**
2. **Setup some dummy data for the List**
3. **Using model & dummy data in a Store**

The data setup portion is pretty vanilla, consisting of a `model` **(1)** that contains fields corresponding to the form elements from listing 8.1, as well as some dummy data **(2)** that fits into the `model` and will be shown in the `ListView` we are about to create. We finish off with a `Store` **(3)** that utilizes the `Model` and the dummy data. The `Panel` that shows our `ListView` is next on the agenda.

Listing 8.8 Creating the List Panel to Launch the Form

```
var myPanel = Ext.create("Ext.Panel", {
    fullscreen  : true,
    layout : 'fit',
    items  : [
        {
            xtype: 'toolbar',
            docked: 'top',
            title: 'Party Invitation Manager'
        },
        {
            xtype: 'list',
            cls:'invite-list',
            store: theListStore,
            itemTpl: '<span class="status status-{inviteStatus}"
title="{inviteStatus}"></span>{lastName}, {firstName}' ,
            listeners : {
                itemtap : showInviteSheet
            }
        }
    ]
});
```

1. **Give the list a class so we can prettify it**
2. **Set a custom template to show extra data**
3. **Hook into the itemtap so we can show the form**

In listing 8.8 we create a basic `Panel` with a toolbar for the title and a "fit" layout, our best option since we only have a single item in the panel that is not docked. We setup a `List` and give it a CSS class **(1)**, which we will use later on to style the list items. In order to show the full name of each invitee along with a status indicator we utilize a custom `itemTpl` **(2)** that contains the necessary fields we need. Last but not least we hook into the `itemtap` event **(3)** of the list, which allows us to show the form to edit list items whenever a list item is clicked.

The `itemtap` handler is relatively simple. It is passed a reference to the `list`, the `index` of the item that was clicked, the target (inside of the DOM) that represents the item on the screen, as well as the underlying record (model instance) for the clicked item. We use the handle we already have on the invitation popup (which we will create next) to load the record into the form using `setRecord` and passing it the `record` provided by the `itemtap` handler. To be extra user-friendly, we lookup the title bar, and give it a new title to reflect the name of the invitee we are editing. In code, that looks like the following:

```
function showInviteSheet(listView, index, target, record, e) {
     invitePopup.down("#invite-form").setRecord(record);
     invitePopup.getComponent('title').setTitle(
        'Edit - ' + record.get('lastName') + ', ' +record.get('firstName')
     );
     invitePopup.showBy(target);
};
```

Last but not least, we need to actually show the panel. For this purpose, we use the `showBy` method, and feed it the `target` of the item that was tapped. This will automatically align the `Panel` with the tapped row, and provide a tiny arrow from the panel to the row, making it especially easy for the user to see which record the panel corresponds to.

The only thing left now is to create the popup that shows the form. For that, we will use an `Ext.Panel` containing a `FormPanel`. Take a peek at Listing 8.9 for the details.

Listing 8.9 Creating the Form Dialog

```
var invitePopup = Ext.create("Ext.Panel", {                          1
     centered: true,
     layout: "fit",
     modal: true,
     hideOnMaskTap: false,
     height: 280,
     width: 310,
     items : [
         {
            xtype: 'toolbar',
            itemId:'title',
            docked: 'top',
            title: 'Edit Invitee'
         },{
            xtype: 'toolbar',
```

```
            docked: 'bottom',
            items: [{
                xtype: 'spacer'                                          2
            },{
                ui: 'confirm',
                text: 'Save',
                handler: function(){                                     3
                    var inviteForm = invitePopup.down('#invite-form');
                    inviteForm.updateRecord(inviteForm.getRecord());      4
                    invitePopup.hide();                                   5
                }
            },{
                ui: 'default',
                text: 'Cancel',
                handler: function(){
                    invitePopup.hide();
                }
            }]
        },{
            xtype  : 'formpanel',
            itemId: 'invite-form',                                       6
            items: [{
                xtype: 'textfield',
                label: "First",
                name: "firstName",
                placeHolder: "Enter First Name Here"
            },{
                xtype: 'textfield',
                label: "Last",
                name: "lastName",
                placeHolder: "Enter Last Name Here"
            },{
                xtype: "selectfield",
                label: "Status",
                name: "inviteStatus",
                placeholder: "nothing",
                options: [
                    { text: 'Undecided', value: 'undecided' },
                    { text: 'Accepted', value: 'accept' },
                    { text: 'Declined', value: 'decline' }
                ]
            }]
        }
    ]
});
```

1. **Use a Panel to show the form in a modal fashion**
2. **Add a spacer to right align the buttons**
3. **Handle saving the form**
4. **Persist form data to the model**
5. **Hide the dialog when done**
6. **Use an itemId for easy access**

Listing 8.9 probably looks rather lengthy, and maybe even a bit intimidating at first glance, but trust me, there is nothing really new here that we haven't covered before. We start out by instantiating an `Ext.Panel` **(1)**, which we assign to the variable `invitePopup`. Remember from our `showInviteSheet` function before, `invitePopup` is the reference we use to talk to the `Panel` and to obtain a reference to the `FormPanel`.

We give the panel a top toolbar for the title, and a bottom toolbar for the save and cancel button. Putting a `spacer` **(2)** as the first item in the bottom toolbar causes the buttons to right align. The save button we wire up to a simple handler **(3)** that obtains a reference to the form by using the down method on the panel in conjunction with the `itemId` of the form. In order to persist the changes from the `FormPanel` to the `Model` instance we used, we utilize the `updateRecord` method, feeding it the record the form is currently bound to **(4)**. The `updateRecord` method takes all the form fields and their current values, and dumps it back into the model. After we're done saving the data, we dismiss the edit dialog by hiding the panel **(5)**. The same is done when the cancel button is pressed. The rest of the code is a plain form with two `textfields`, and a `SelectField`. The only notable thing about the form is that we are giving it an `itemId` so we can easily obtain a reference to the form using the `down` method from the `Sheet`.

If you run the code as we have it right now, you should get a list that shows the 3 dummy items, and allows you to edit each item. You might wonder why you're not seeing the colored dots, like figure 8.8 illustrates. D'uh, we forgot to style the list of course. Just add the following styles to the page, and we're good to go.

```
.invite-list .status {
    display: inline-block;
    vertical-align: middle;
    width: 20px;
    height: 20px;
    border-radius: 10px;
    margin-right: 10px;
}

.invite-list .status-decline { background-color: red; }

.invite-list .status-undecided { background-color: grey; }

.invite-list .status-accept { background-color: green; }
```

This concludes the Party Invitation Manager sample. Before we wrap up, let's address two potential questions you might have. If you're wondering what you would do if your form had more than three fields in it, in which case the floating panel might be too small, then the answer is pretty simple. Don't use a floating Panel! Go with a Panel and a card layout instead. The first card would contain the list, and the second card the form. Whenever a user clicks a list item, you would follow the same process we currently have, but instead of showing the floating panel, you would tell the card layout to navigate to the card with the

form. On the flipside, the form would tell the card layout to navigate back whenever the user saves data, or cancels.

The other thing you might wonder is why we setup the form in advance, instead of creating a new one every time a user clicks a list entry. The main reason here is that the single form conserves processing time, since it only needs to be created once, and can then be reused over and over again. This is an important paradigm to remember, as mobile devices are more restricted on system resources. In an ideal world, we would take this principle even further by not creating the form in advance, but instead have a check in the `showInviteeSheet` function that checks whether the form has been created yet or not, and if not, goes of and creates it. This would address the use case where a user only wants to see the list, but doesn't make any edits.

8.7 Summary

In focusing on the `FormPanel` class, we've covered quite a few topics, including many of the commonly used fields, ways to arrange forms in creative ways, and most importantly, how to deal with form data. Although forms in Sencha Touch still need to mature a bit, and could certainly use a few more helper functions to make everyone's life easier, a lot of that is just a matter of time while the community adds plugins & extension, and code from Ext4 gets ported over to Sencha Touch. Until that happens however, you will have to jump through a few extra hoops to accomplish the same thing without the helpers.

Moving forward, in the next chapter we will be taking a look at the exciting world of maps & media, before finally topping it all off with a larger application.

9 Maps and Media

This chapter covers

- How to use Sencha Touch's Map Component
- Looking at using the Image Component
- Exploring multimedia with the Video and Audio components

So far in this book we have covered how to create and layout components and even retrieve data from a backend to populate `DataView` and `List`. Feels pretty good to get this far doesn't it? Let's push us a little bit further and see what we can do with Sencha Touch and modern mobile devices. In this chapter we will look at `Maps` and how to use the Google Maps API, dive into the simple `Img` (image) component and spend some time with the `Media` component and its subclasses `Audio` and `Video`. First up, the `Map` component.

9.1 Maps in your application

Quite often I see Google Maps used in mobile applications. Have a Contact Us section of your application? Could be a nice decision to have a map to show your users the location of your business. Figure 9.1 shows an application that uses Google Map markers to show house listings in the Houston, Texas area but what you may not know is how surprisingly easy the functionality to add markers actually is. You may think that is hard but with Sencha Touch's `Map` component and Google Maps and its API it is actually not that hard. In this section we will first take a look at the Sencha Touch `Map` component, create a simple map and dive into using the Google Maps API to add markers.

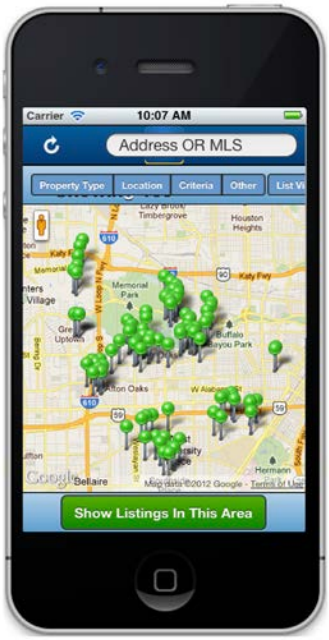

Figure 9.1 Example of a map using markers for points of interest. This application was created by the Houston Association of Realtors to show housing listings.

9.1.1 Maps under the hood

First looking at what `Ext.Map` does under the hood can get confusing very fast. You have the Sencha Touch `Map` (`Ext.Map`) component but you also have the Google Map component making it hard to understand what map component you actually need to use. To make it worse `Ext.Map` also uses `Ext.util.Geolocation`, which is a utility class to get GPS coordinates and some other useful information using the HTML5 Geolocation spec. So as you can see it may be confusing but we will take it slow and discuss each one at a time and then bring them together at the end.

9.1.2 Location aware

When the W3C was working on the HTML5 family, one of the highly touted specs was the Geolocation spec. This spec gave rules on how devices should allow the browser get the current location. Sencha Touch has a utility class called `Ext.util.Geolocation` that takes advantage of this spec and allows you to poll to get location updates.

W3C Geolocation

You can learn more about this spec directly from the W3C's website. It's a very technical description but allows you to truly understand what is going on in the background. The URL address is:

http://dev.w3.org/geo/api/spec-source.html

A good thing to know is that the Geolocation spec does require the browser to ask for permission to get the location. The first time you try to get the location, the browser will prompt your user for permission to allow the application to receive location updates. This setting will be saved so it won't keep bothering the user each time you get location. Let's dive into an example of how to use the `Ext.util.Geolocation` class.

Listing 9.1 Using the Geolocation class

```
var geo = new Ext.util.Geolocation({
    allowHighAccuracy : true,                                    // 1
    listeners         : {
        locationupdate : function(geo) {                         // 2
            console.log('New latitude: ' + geo.getLatitude());
        },
        locationerror  : function(geo, timeout) {                // 3
            if (timeout){
                console.log('Timeout occurred.');
            } else {
                console.log('Error occurred.');
            }
        }
    }
});
geo.updateLocation();                                            // 4
```

{1} Getting high accuracy with the allowHighAccuracy config
{2} Listen for the locationupdate event
{3} Error handling via the locationerror event
{4} Start the polling

Listing 9.1 shows how easy it is to use the Sencha Touch `Geolocation` class. By default, the location's accuracy may not be that accurate, it would be good for general location but if you need more accuracy then you need to set the `allowHighAccuracy` config to `true` #1. Do be aware that this config has adverse affects on the time it takes to get a location and may use more of the device's battery. Getting location is asynchronous so we have to listen for the `locationupdate` #2 event to handle when a location is finally retrieved and the `locationerror` #3 event as sometimes it may fail. In the `locationupdate` event you can get the latitude and longitude coordinates among other information about the current location. The `locationerror` can fire either the timed out due to not being able to connect to the GPS satellites or just because something went wrong. Just creating the `Geolocation` class nothing will happen so we have to execute the `updateLocation` #4 method. You have

to execute this method each time you want to get a new location but by default the autoUpdate config is set to true so it will automatically poll and fire the locationupdate event. The frequency of this automatic update is based on the frequency config, which is 10000 (10,000ms), by default.

Now that we know how to get our location, let's look at how we can create a map.

9.1.3 Creating a simple Map

We need to cover just a few quick things before looking at some code. In your index.html you will need to add a <script> tag to load the Google Maps API, without this nothing will work and you will receive some error messages. This is an example of what this should look like:

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

Let's look at Listing 9.2 to see how to create a Map component.

Listing 9.2 Building a simple Map

```
new Ext.Map({
    useCurrentLocation : true,                                    // 1
    mapOptions         : {                                        // 2
        mapTypeControl  : false
    }
});
```

{1} Use current location to center map
{2} Configure the Google Maps instance

That's it! Just six lines of code and we have a map that uses our GPS location. Technically we don't need to have any configuration options set and we would still have a map rendered for us. However, we set the useCurrentLocation #1 to true so that we could use Ext.util.Geolocation to determine our current location and then center the map on that location. If useCurrentLocation is set to true, the map will not render until the location is found. We then specified the mapOptions #2 configuration to an object to configure the Google Maps instance. We set the mapTypeControl option here to not allow the user to switch between the different map types that would otherwise be available like a satellite view instead of the road view. You can view more configuration options that can go into this mapOptions configuration by going to http://code.google.com/apis/maps/documentation/javascript/reference.html#MapOptions.

Figure 9.1 shows the result of the code from Listing 9.2

Figure 9.2 End result from code from Listing 9.2 to show a simple Map component in action.

9.1.4 Getting advanced with Google Maps API

So we just learned how to create a simple map. So far so good, but now let's get a little more advanced and learn how to incorporate more features from the Google Maps API. We won't get too in depth so we will just learn how to add a marker to the map and show an InfoWindow to show some information on what that marker is there for. Starting off, let's get the foundation created by creating a simple `Ext.Map` instance in Listing 9.3:

Listing 9.3 Create a simple Map to start

```
var onMapRender = function() {},
    map         = new Ext.Map({
        fullscreen         : true,
        autoUpdate         : false,                    // 1
        useCurrentLocation : true,                     // 2
        listeners          : {
            maprender : onMapRender,                   // 3
            single    : true                           // 4
        }
    });
```

{1} Disable automatic updating
{2} Start centered on current location
{3} Wait for the map to be rendered
{4} Remove the listener using the single event option

In this example we do not need the automatic updating of our position so we set autoUpdate to false #1. If we do not need the automatic updating but left this to be true we would be wasting CPU cycles, memory and battery by polling the current location every 10,000 miliseconds and we don't want to waste performance and the user's battery. We want the map to render centered on our current location so we set the useCurrentLocation to true #2. Finally we can't add any makers to a map that isn't rendered so we have to listen for the maprender event #3 and after its first firing it will remove the listener by using the single event option #4. The maprender event is fired on Ext.Map when the Google Map is inserted and rendered. Just like we set autoUpdate to false not to waste resources we can remove the maprender listener, as we know it's only going to fire once therefore saving some memory.

So we have a map ready but it's just a plain map and we want a Google Marker to show on the map where we are. The Google Marker also gives some context to the map, without the Marker we would just have a plain map and your users may not know what they are really looking at. In Listing 9.3 we created a simple map and added a listener to the maprender event but the onMapRender function is an empty function and we need to take action within that function to show the marker. What we want to accomplish is to have a marker render where our current location is and when a user clicks, or taps, on that marker have a Google InfoWindow appear to let the user know what that marker is there for.

Listing 9.4 Create a Marker and InfoWindow

```
var onMapRender = function(STmap, map) {
    var maps   = google.maps,
        geo    = STmap.getGeo(),                                    // 1
        lat    = geo.getLatitude(),
        lng    = geo.getLongitude(),
        coords = new maps.LatLng(lat, lng),                         // 2
        marker = new maps.Marker({                                  // 3
            position : coords,
            map      : map
        }),
        infowindow = new maps.InfoWindow({                          // 4
            content : 'This is the current location'
        });

    maps.event.addListener(marker, 'click', function() {            // 5
        infowindow.open(map, marker);                               // 6
    });
};
```

{1} Get the Geolocation class instance
{2} Create the Google LatLng instance
{3} Create the Google Marker instance
{4} Create the Google InfoWindow instance
{5} Add a click event listener to the Marker
{6} Open the InfoWindow

In Listing 9.4 we updated the `onMapRender` function that gets fired from the `maprender` event as shown in Listing 9.3. First we cache the `google.maps` namespace onto the `maps` variable just for minification and very tiny, minimal performance gain purposes. We then need to get the location so we use `getGeo` #1 on the `STmap` argument, which is the `Ext.Map` instance and returns the `Geolocation` instance. To get the latitude and longitude we execute the `getLatitude` and `getLongitude` methods to retrieve those coordinates from the `Geolocation` class. We need to create a Google LatLng instance #2 based on the latitude and longitude coordinates so the marker knows where to show in the Google Map. Finally the Marker is created #3 passing in the LatLng instance as the position config and the Google Map instance as the `map` config. If we were to stop there we would have a Marker show on the map but we also wanted an InfoWindow to describe the Marker. So we then create the InfoWindow instance #4 passing in some content which can hold HTML but be careful of how much you put in there as the user may be on a cell phone that has limited screen realestate. We then add a `click` listener on the marker with the `addListener` method #5 so that we can open the InfoWindow using the open method #6 passing in the Google Map instance and the marker to show next to. View Figure 9.2 to see what Listing 9.4 yields.

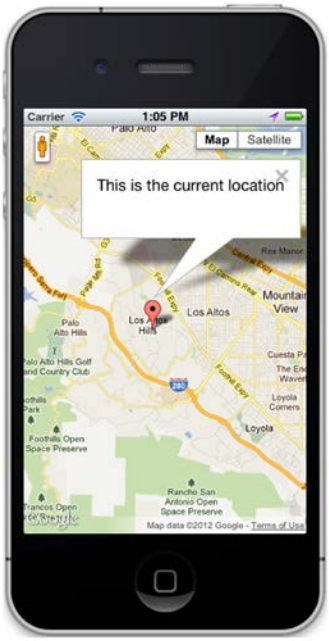

Figure 9.3 Google Marker added to the Map. Tap on the marker to view the Google InfoWindow.

That's it! You should now have a general understanding on how to incorporate the Google Maps API with the Sencha Touch Map Component but there is so much you can do with the Google Maps API. We have only used Marker and InfoWindow but you can show directions between two points or get information about the elevation of a certain coordinate. Referencing the Google Maps API documentation you can see the different classes you can use and Google even has some samples. Next up, some fun with the `Img` Component!

9.2 Handling images

You may be thinking, "Why is there an image component within Sencha Touch? It's so basic." I would agree. Images are such a basic thing that there may not be a need. The reason Sencha Touch has an `Img` component is to provide a way to show an image from a simple configuration and also provide a load and error event to let you know when an image is loaded or when loading has failed. For the `Img` component, getting an image to play nice in layouts would require some extra steps and Sencha Touch has removed those extra steps and also added some additional features that you could use to create some stunning things.

9.2.1 Image basics

The `Img` component simplifies the job of having images be held in a layout but also complicates the issue, as it's not just a simple Component setting an image as the html of the Component. There are two modes to the `Img` component, background and image. The background mode simply sets the `background-image` CSS rule to the URL of the image you pass in on the `<div>` element. The image mode will create a `<img>` tag and set the `src` attribute to the URL of the image you pass in and is wrapped in the `<div>` element. So you can tell the two modes may accomplish the same thing but go at it two different ways. Here is a simple way to define the `Img` component:

```
new Ext.Img({
    width  : 100,
    height : 100,
    src    : 'path/to/image.png'
});
```

By default the `Img` component is in background mode so if you were to inspect the DOM you would notice a single `<div>` element with the `background-image` CSS rule set to the `src` config. Now we can look at the image mode:

```
new Ext.Img({
    width  : 100,
    height : 100,
    src    : 'path/to/image.png',
    mode   : 'image'
});
```

Figure 9.4 Visually, both mode configurations look the same; background mode on left and image mode on right. Check out the DOM and each Img component will look different.

Now if we were to inspect the DOM we would see an `<img>` element wrapped in a `<div>` element but if you compare the two end results you may not see a difference. Let's remove the `width` and `height` configs from both instances and check the end result, notice that the image mode has a size but the background mode `Img` component does not. Why is that? It's because the `<img>` tag is telling the `<div>` that it has a size and due to the styles on the `<div>` it is allowing itself to be autosized based on the size of the `<img>` element where as the background mode `Img` component the `<div>` element can't get a size from its CSS `background-image` rule.

More advanced users will also note that you can do different things with each DOM structure. Using the default background mode you can tile the image so the image will repeat itself horizontally and vertically or just one direction using CSS. Also with CSS you can simply center the background image where if you used the image mode you would have to use more CSS to get the center effect.

Moving forward, you have other functionality like the `Img` component exposes a load and error event to notify you if the actual image was loaded or if there was an error along with a tap event that will fire when it detects it has been tapped on. You can also use the `setSrc` method to change the URL of the image on-the-fly and even use the `setMode` to change between background and image mode on-the-fly. Another hidden gem of the `Img` component

is that when it is hidden, the image is removed from the DOM but since the browser will cache the image when the Img component is shown again it will reset the src attribute and the image will appear instantly. This will free up some memory, as the image doesn't need to be in memory when hidden, a great feature of Sencha Touch 2.

9.2.2 Pre-loading an image with a spinner

Showing a static image with the Img component is simple and powerful. Let's do something fun and use what we learned so far in this book. Let's build a proof-of-concept example of showing a pre-loading spinner while the image we want to show loads. What we are going to do is have a Container have two Img components. The first item of the Container will be the Pre-loading spinner Img component that has been Base64 encoded so that it doesn't have to be downloaded but can be shown right away. The second item will be the actual image we want to show but may take a second or two or more to download on a mobile cellular network that usually has high latency. The Base64 encoded spinner image won't be shown here as it's quite a long string, this image comes with Sencha Touch and is located at /sencha-touch-2.0.0/resources/themes/images/default/loading.gif.

Listing 9.5 Image Pre-loading spinner example

```
new Ext.Container({                                                    // 1
    fullscreen : true,
    items      : [
        {
            xtype  : 'image',                                          // 2
            height : 250,
            width  : 250,
            cls    : 'loading-image',
            src    : '-------'                   // 3
        },
        {
            xtype     : 'image',                                       // 4
            height    : 250,
            width     : 250,
            hidden    : true,                                          // 5
            src       : 'path/to/image.png',                           // 6
            listeners : {
                delay  : 2000,                                         // 7
                load   : function(image) {
                    var container = image.up('container'),             // 8
                        spinner   = container.down('image');           // 9

                    container.remove(spinner);                         // 10
                    image.show();                                      // 11
                }
            }
        }
    ]
});
```

{1} Create a Container to wrap the Image Components
{2} Pre-loading spinner Image Component
{3} The Base64 string of the spinner image

{4} The actual Image Component
{5} Actual Image Component is hidden
{6} The URL of the image
{7} Delay the load event
{8} Resolve the Container
{9} Resolve the Pre-loading spinner Image Component
{10} Remove the Pre-loading spinner Image Component
{11} Show the actual Image Component

In Listing 9.5 we use a `Container` #1 to wrap the two `Img` components. What we are actually doing here is having two image items under the `Container` and once the actual image is loaded we will remove the loading image and show the actual image. The first item is the `Img` component #2 for the pre-loading spinner image that uses the Base64 encoded string #3 of the loading.gif animated GIF image that comes with Sencha Touch 2. The second item is the actual image #4 that we want to show but since we want control over when that `Img` component is shown we have it marked as hidden #5. Of course we use a URL #6 to load the image from. Also notice that we set the height and width of both `Img` components; we did this so that the size will remain the same when we switch to the actual `Img` component in the `load` event listener.

Since we are only creating a proof-of-concept example right now we are using a computer and that means we have a reliable Internet connection so to see if it all works we can use the `delay` event option #7 to delay the firing of the `load` event listener by 2,000 miliseconds. Within the `load` event listener we get the `Img` component instance as the first argument so we can resolve the `Container` by using the `ComponentQuery` convenient method up #8 on the `Img` component instance. Now that we have the `Container` instance we can get the pre-loading spinner `Img` component by using the `ComponentQuery` convenient method down #9 on the `Container` and since the pre-loading spinner `Img` component is the first child item of the `Container` and the `down` method will return the first instance it finds so we know it will return the correct `Img` component we want. Since the `load` event fired on the actual `Img` component we don't need to pre-loading spinner `Img` component so we can go ahead and remove #10 the pre-loading spinner `Img` component so that the memory it is using can be used for other things. Lastly we show the actual `Img` component #11.

So run the code replacing the `src` configs with the Base64 string and a URL to a remote image and possibly updating the width and height configs. For a remote image URL I found a nice Sencha logo at http://www.sencha.com/files/blog/old/blog/wp-content/uploads/2010/06/sencha-logo.png, which is 250px by 250px. Works pretty good huh?

Figure 9.4 will show the before and after the image loading from this test.

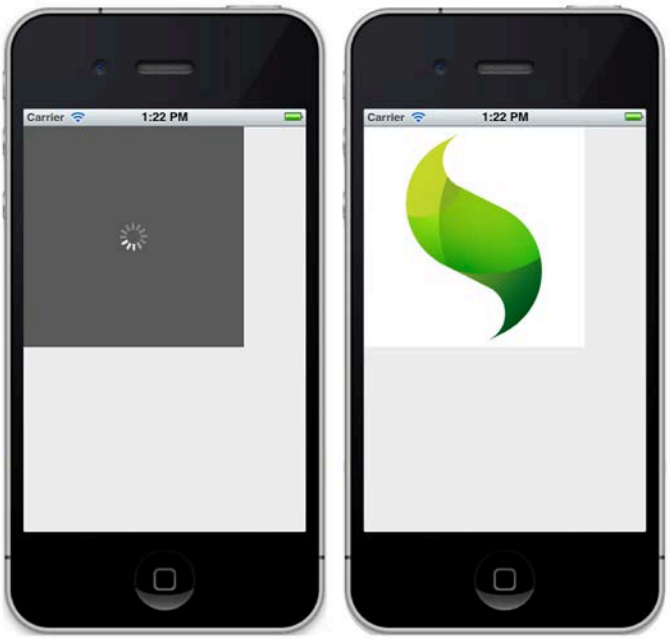

Figure 9.5 Shows the loading spinner (left) and the actual image loaded (right)

This pre-loading spinner is a great candidate to be created into its own class to handle this internally. To do this you would use `Ext.define` to define a new class, which will be covered in Chapter 10.

9.3 Mastering Media

Another feature of an application that could be a large selling point is adding multimedia, audio and video. Maybe you have an application that helps teach a user a subject. Reading text will work but playing audio or even better playing a video to walk the user through something is a great tool to have and your users would most likely keep coming back because of this multimedia experience. In this section we will be exploring both the `Audio` and `Video` components, but first, we need to cover the base class, `Media`.

9.3.1 Media base

As we learned back in Chapter 3, Sencha Touch has this Component Model where components extend others to introduce common functionality among subclasses (call abstraction) and dependability. The `Audio` and `Video` components both extend from the `Media` component. So what does the `Media` component give the `Audio` and `Video` components?

First let's understand what is going on under the hood. HTML5 has both an <audio> and <video> DOM element and there are many common features of these tags like common events and ways to control them. The Media component is a wrapper of these so that its subclasses can have this functionality without duplicating the code. Now we know an overview of what is going on under the hood, let's take a look at a little more detail of the common functionality in the Media component.

Events are usually a large part of your app so having some useful events is key. Think of a DVD player; it plays, pauses, stops, rewinds and fast-forwards, basic functionality that every model must have. The Media component bubbles play, pause, ended and timeupdate events from the underlying <audio> or <video> so you have events for the basic functionality that a Media component should have as shown in Figure 9.5. We also have a volumechange event that will fire when you use the HTML5 controls to change the volume. Along with the volumechange event, you also get a mutedchange event that will fire when the <audio> or <video> element is muted or unmuted. There are quite a few useful events to build multimedia functionality into your application.

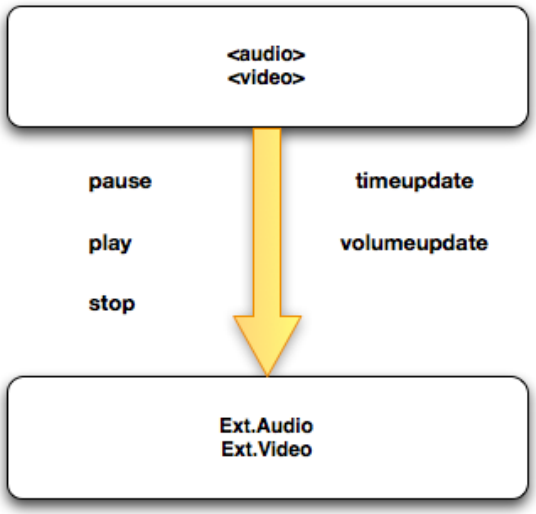

Figure 9.6 Shows the 5 events that get bubbled from the <audio> and <video> DOM elements to the Ext.Audio and Ext.Video Sencha Touch components.

NOTE The volumechange event will not fire if you change your device's volume. This event will only fire if you use the on-screen control to change the volume of the media or use the setVolume method.

Table 9.1 shows some of the most used configuration settings.

Table 9.1 Describing the common configurations.

Config	Description	Defaults
url	The URL of the media file to be used.	Empty string, ''
enableControls	true to show the on-screen controls to play/pause, seek and change the volume.	Android – false Others – true
autoResume	true to start playback when the Component has been visually activated.	false
autoPause	true to stop playback when the Component has been visually deactivate.	true
preload	True to begin the preloading of the media immediately.	true
Loop	true to loop the media forever.	false
volume	The volume level of the media accepting a value between 0 (no volume) and 1 (full volume).	1
muted	true to have the media volume to be muted.	false

The most used config will be the `url` config so that we can play a media file. If you don't want controls on screen so that you can change the volume or seek through the media file, then you can set the `enableControls` to false which defaults to `true` unless you are on Android, in that case it will default to false. If you want the media to start playing when the `Media` component is visually activated then you can set the `autoResume` to `true`, defaults to false. Users may not want to wait around for media and may decide to jump to another section of your application without stopping the playing of the media, the `autoPause` config which defaults to `true`, will pause the media when the `Media` component has been visually deactivated saving quite a bit of resources on the device. You also have a `loop` config that will loop the media when the media reaches the end, defaults to `true`. To configure the volume you have a `muted` config that can start the media muted, defaults to false, and you also have a `volume` config that accepts a value from zero to one defaulting to one. You can visit Sencha Touch 2's API documentation for these configuration settings and more. Setting up the `Media` component and listening for it's events just two-thirds of the functionality, let's look at some of the methods to control the `Media` component programmatically.

Table 9.2 shows some of the most used configuration settings.

Table 9.2 Describing the associated get and set methods to the configurations from Table 9.1.

Config	Get Method	Set Method	Returns/Accepts
url	getUrl	setUrl	String
enableControls	getEnableControls	setEnableControls	Boolean
autoResume	getAutoResume	setAutoResume	Boolean
autoPause	getAutoPause	setAutoPause	Boolean
preload	getPreload	setPreload	Boolean
Loop	getLoop	setLoop	Boolean
volume	getVolume	setVolume	Int (0-1)
muted	getMuted	setMuted	Boolean

Each of the configurations has a set method associated with the first letter of the config upper-cased and 'set' prefixed. For example you can change the url on-the-fly with the `setUrl` method. If the media was playing at the time the `setUrl` method fires, it will start the playing of the new media. You also get the `setEnableControls`, `setAutoResume`, `setAutoPause`, `setLoop`, `setMuted` and `setVolume` methods from the configs. With these alone you can see how much control you now have over the media. That's not all the control you have, you also have `play`, `pause` and `stop` methods and also a `toggle` method. The `toggle` method toggles the playback status between playing and paused. Another method to control the media is the `setCurrentTime` method that allows you to jump around the timeline of the media.

Not all methods are used to control the media; you also have methods to get information about the media. These methods are `getDuration`, `getCurrentTime` and `isPlaying`. `getDuration` will tell you how long the media is in seconds. `getCurrentTime` will tell you where in the media the current time is in seconds. To find out is the media is currently playing you can use the `isPlaying` method, which will return a Boolean whether or not the media is playing. Like the set methods from the configs, you also have the get methods from the configs: `getUrl`, `getEnableControls`, `getAutoResume`, `getAutoPause`, `getLoop`, `getMuted` and `getVolume`.

Wow, that's a lot of functionality but remember, this is just the base class of the `Audio` and `Video` components so each of them have all of these configs, events and methods providing a lot of control over your media. We just covered the `Media` component, now let's look at its first subclass: `Audio`.

9.3.2 Listening to audio

Since the `Media` component has all that functionality, the `Audio` component doesn't actually have any more functions to provide. The `Audio` component is basically just a

configuration Component to setup the DOM elements that the functionality from the Media component can control. We have been talking a lot so let's just jump into the code. Listing 9.6 is an example of how to setup an Audio component:

Listing 9.6 Simple Audio example

```
new Ext.Audio({
    fullscreen : true,
    url        : 'assets/audio/audio.mp3',                    // 1
    autoResume : true,                                        // 2
    loop       : true,                                        // 3
    volume     : 0.5                                          // 4
});
```

{1} The url of the audio file
{2} Resume playback on activation
{3} Loop the audio endlessly
{4} Half volume

In Listing 9.6, you can tell setting up an Audio component is simple. We gave a url #1 to an MP3 file to play. When the Audio component is visually activated, we want to resume setting autoResume to true #2. We don't want the audio to stop playing so we set loop #3 to true so that it will just continue to play instead of stopping at the end. You never know how loud the user has his volume set so setting the volume configuration #4 to half volume or even less may be a good idea as the user could change this at any time.

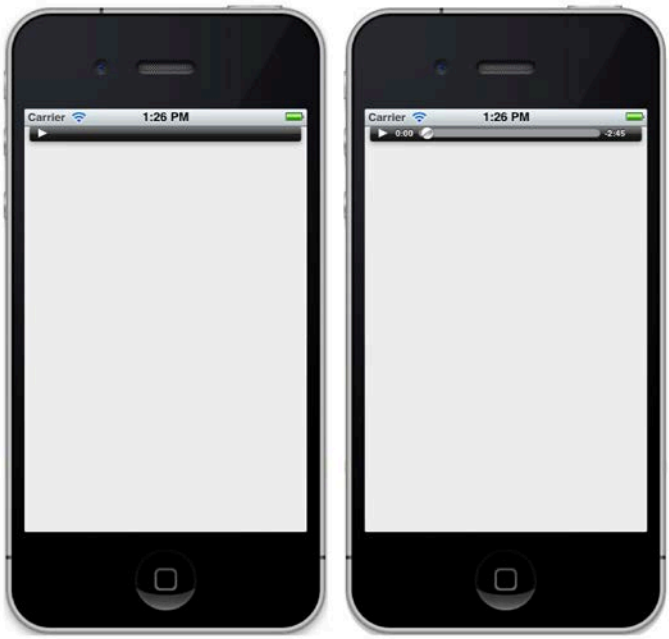

Figure 9.7 The Audio views of before playback (left) and after press the play button on the on-screen controls (right)

To be a little more advanced you could put the `Audio` component within a `Container` and give the `Container` a docked `Toolbar` with some buttons to control the playback and volume as shown in Figure 9.7 using the knowledge we know about the control methods that the `Media` component brings. Once we are done talking about the `Video` component, we can look at some of these functions.

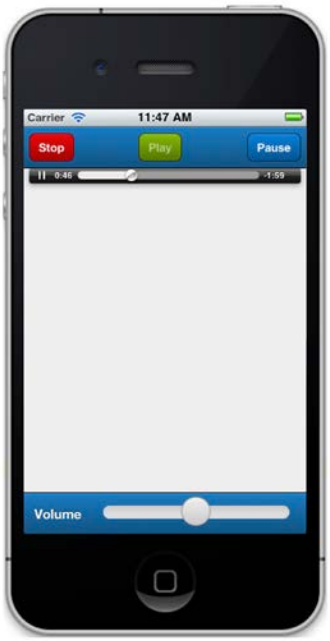

Figure 9.8 Shows an example of having an Audio component in a Container with some toolbars to handle simple playback and volume change.

There is one thing to note about using the `Audio` component; it can only work if the device, more the operating system, can play the audio file. Not every audio file and codec is support on a mobile operating system. For instance, iOS and Android do not support the Windows Media formats like WMA. The recommended formats for cross-device compatibility is uncompressed WAV and AIF, MP3, AAC-LC, and HE-AAC with MP3 probably going to be the most common format supported. If the Audio Component isn't playing, more than likely the codec is not supported and trying one of the other recommended formats could be a quick test in order to debug the issue. You would be amazed of how many times someone has complained that the `Audio` component isn't working and it was because the device's volume was turned down or muted.

Since this is only audio, you won't see anything except for the on-screen controls unless you specified the `enableControls` config to false. Even if the MP3 file has album artwork, the HTML5 audio will not display this. You can get around this by using a `Container` with an `Img` component to display an image and an `Audio` component as children of the `Container`. Just having the `Audio` with its on-screen controls may look a bit underwhelming to your user but this is up to your UI.

That's the basics of the Audio component. Sencha Touch makes playing audio very simple. Hopefully your mind is going crazy thinking of all the audio stuff you can do in your application. Let's kick it up another gear and look at playing video.

9.3.3 Playing video

Like the Audio component, the Video component extends the Media component so Video gets all the methods, configurations and events. The Video component adds one extra feature and that is the use of a poster. A poster is an initial image that is used instead of showing the video; tapping on the poster will start playing the video. This is a departure from the <video> tag specification but the reason Sencha Touch does this is due to device resource limitation. It takes quite a lot of resources to handle video, as the device has to decode the video file and then playing it. Scrolling around an application would feel very choppy and not provide your users a great experience. To get around this, Sencha Touch shows a simple image (poster) in place of the video allowing for application performance to be much better. Of course once that video is playing, the benefits of the poster will be lost. Let's take a look at creating a Video instance

Listing 9.7 Simple Video example

```
new Ext.Video({
    fullscreen : true,
    url        : 'assets/audio/video.mp4',                    // 1
    posterUrl  : 'assets/poster/video.png',                   // 2
    autoResume : true,                                        // 3
    loop       : true,                                        // 4
    volume     : 0.5                                          // 5
});
```

{1} The url of the video file
{2} The poster image url
{3} Resume playback on activation
{4} Loop the video endlessly
{5} Half volume

This will seem very familiar to Listing 9.6 when we constructed an Audio instance. We first gave the Video a url #1 of an MP4 video file and also a url to a PNG image to serve as the poster #2. To keep with Listing 9.6, we used the autoResume #3 to resume video playback on activation, loop #4 to keep the video playing endlessly and set the volume #5 to only half so that we do not play video with loud sound.

Figure 9.9 Showing the Video component on an iPhone with the poster (left) and in the native video player when playing the video (right).

Once again, not every mobile operating system can play every video file format and codec. What does all this mean to us, web developers? It means there really is no standard video format in the HTML5 video spec. Both iOS 5.1 and Android 4 currently support an MP4 file with H.264 at the time of writing; however, Google has pulled H.264 support in its popular Chrome browser for personal computers so the future of H.264 support in Android is unknown. Google has made no announcement at the time of writing that it will pull H.264 support in Android. I would be surprised if iOS stops H.264 support anytime soon even if a competing codec is adopted in the HTML5 video spec; at that point I think it is safe to bet Apple will continue to support H.264 and the new codec. I really don't see all companies setting politics behind and agree on a common codec each would be happy on even though it would be for the good of the web.

HTML 5 Video History

At first Ogg Theora was recommended as the video codec to be used as a standard but there are some unknown patents that have scared large companies like Apple from supporting it. Interestingly, the H.264 codec may also have unknown patents but H.264 has been widely used so the threat to mobile operating system makers has been

decreased to almost nothing. Even though H.264/MPEG-4 AVC has been wildly used but has not been made part of the HTML5 video specification due to the fact that users have to pay licensing fees to the MPEG LA group; Microsoft and Apple are among the members of this group.

iOS is made by Apple, a MPEG LA group member, so Apple really doesn't have to worry about these licensing fees as much as Google, the founder of the Android operating system. Google is not part of the MPEG LA group but still supports the H.264 codec. Google acquired On2 and therefore got the VP8 codec and provided a royalty-free licensing to users of the VP8 codec. Google combine the VP8 video codec with the Vorbis audio format within a Matroska container creating the WebM format. Microsoft and others have criticized the WebM format but Mozilla and Opera have asked for the inclusion of WebM into the HTML5 video specification.

Figure 9.10 Showing the Video component running on an iPad where you can see the poster (left) and the video is playing inline in the application (right)

9.3.3 Thing to keep in mind

We need to go over a few things about using media in your application. We need to remember that we are on a mobile network that has high latency and is not as reliable as home broadband. Having many `Audio` or `Video` components that are loading at the same time will choke the connection on your user's device. You have no way in knowing what else they will be doing or where they are. Your users could be on a network that isn't great and connection is slow so loading audio and video files may take forever so loading one at a time and optimizing those files for mobile devices is recommended. For instance, a video file that looks great on a high-definition, 1080P television is way too much for a mobile device. You can keep 1080P high-definition but the video size should be smaller as tablets and phones don't have large screens so it would be a waste to have a large sized video be used.

Performance is another issue. Today's devices are more powerful than ever having more memory, even quad-core CPUs and GPUs and mobile connections that rival home broadband connections. However, even these mobile devices can still be strained with large video files and you can't always be certain that your users will have the latest and greatest devices. A good application will keep performance in mind with every feature implemented and be tested on older devices. I still see iPhone 3 devices view my applications so I still need to support these devices. As long as you put performance first, you will have no issue in finding ways to still have every feature you want.

Using poster images for videos will help performance quite a bit. Scrolling with inline videos without a poster image will provide a bad experience by looking like your application has poor performance. The user doesn't know that the device is actually having problems keep everything rendered at once. Using a poster image, the device doesn't have to try to keep videos rendered and keep up with the scrolling; images are tremendously smaller strain on the device.

One more thing to know is each device is going to handle videos differently. The iPad will play the video inline with your application (depicted in Figure 9.7) as you would want but the iPhone with the same iOS version will launch the video in the native video player (depicted in Figure 9.6) so the video is not actually playing within your application. This really is not going to affect you and your application as when the user presses the Done button it will go back to your application where you left off but just something to know as this is the behavior of the device not your application and what your users will experience. I suggest trying to view your application on a variety of devices so you know the experience your users will have and possibly make changes based on your experience and what you want your users to experience while using your application.

9.4 Summary

This chapter added a considerable amount of fun to your applications. We first looked at how the `Map` component works and learned about how to incorporate the Google Maps API. The pair can be used together to give your application a great user experience. We then looked at the `Img` component and looked under the hood to realize that the `Img` component is much more than meets the eye. We created a proof of concept of using a pre-loading spinner image while the real image loads. Lastly we dove into the `Media` and its subclasses `Audio` and `Video`. We learned that the `Media` class has a lot of functionality that the `Audio` and `Video` components inherit. We saw how to create and setup an `Audio` and `Video` instance and also what codecs can be used. Learning about the history of the HTML5 video spec allows us to be intelligent while developing our application and also learned that playing videos can have consequences on the device and its mobile connection. By using what we learned in this chapter, the user experience of your application can improve by adding some additional map and media functionality.

In the next chapter, we will look at how to spec and develop an application using what we have learned thus far.

10

Class system foundations

This chapter covers

- Understanding the basics of JavaScript prototypal inheritance
- Developing your first extension
- Getting to know how plug-ins work
- Developing a real-world plug-in
- Looking into the Sencha Touch Class loader

Every Sencha Touch developer faces the challenge where reusability becomes an issue. Often times, a component of an application is required to appear more than once within the application's usage lifetime. Without mastering these techniques, you could end up with what is known as "function soup", which is unmaintainable code. This is why we'll focus on the concept of reusability with the use of framework extensions and plug-ins.

In the first section of this chapter, you'll learn the basics of extending (subclassing) with Sencha Touch. You'll begin by learning how to create subclasses with JavaScript, where you'll get to see what it takes to get the job done with the native language tools. This will give you the necessary foundation to refactor your newly created subclass to leverage the Sencha Touch Class system.

Once you've gotten familiar with creating basic subclasses, we'll focus our attention on extending Sencha Touch `Components`. This is where you'll get to have fun learning about the basics of framework extensions and solve a real-world problem by extending the `gridPanel` widget and seeing it in action.

When we finish the extension, you'll learn how extensions solve problems but can create inheritance issues where similar functionality is desired across multiple widgets. Once you understand the basic limitations of extensions, you'll convert your extension into a plug-in,

where its functionality can easily be shared across the `grid Panel` and any descendant thereof.

After we have a solid foundation in with the Sencha Touch class system, we're going to take a few moments to look at the dynamic class loader that Sencha Touch provides. Here you'll learn the popular three patterns for using the dynamic class loader and the caveats for each one.

This is going to be a very fun chapter!

10.1 Classic JavaScript inheritance

JavaScript provides all of the necessary tools for prototypal inheritance, but it falls short in the area to easily setup multiple class inheritance. Sencha Touch makes multiple class inheritance much easier with the class system. To begin learning about inheritance, you'll create a base class.

To help you along, envision we're working for an automobile dealership that sells two types of car. First is the base car, which serves as a foundation to construct the premium model. Instead of using 3-D models to describe the two car models, we'll use JavaScript classes.

> NOTE If you're new to object-oriented JavaScript or are feeling a bit rusty, the Mozilla foundation has an excellent article to bring you up to speed or polish your skills. You can find it at the following URL: https://developer. mozilla.org/en/Introduction to Object-Oriented JavaScript.

We'll begin by constructing a class to describe the base car, as shown in the following listing.

Listing 10.1 Constructing our base class

```
var BaseCar = function(config) {                                    //1
    this.octaneRequired = 86;
    this.shiftTo = function(gear) {
        this.gear = gear;
    };
    this.shiftTo('park');
};
BaseCar.prototype = {                                               //2
    engine    : 'I4',
    turbo     : false,
    wheels    : 'basic',
    getEngine : function() {
        return this.engine;
    },
    drive     : function() {
        return 'Vrrrrooooooom - I'm driving!';
    }
};
```

#1 Create constructor
#2 Assign prototype object

In listing 10.1, you create the `BaseCar` class constructor #1, which when instantiated sets the instance's local `this.octaneRequired` property, adds a `this.shiftTo` method, and calls it, setting the local `this.gear` property to `'park'`. Next, you configure the `BaseCar`'s `prototype` object #2, which contains three properties that describe the `BaseCar` and two methods.

You could use the following code to instantiate an instance of `BaseCar` and inspect its contents with The webkit javascript console:

```
var mySlowCar = new BaseCar();
mySlowCar.drive();
console.log(mySlowCar.getEngine());
console.log('mySlowCar instance:', mySlowCar);
```

Figure 10.1 shows what the output of this code looks like in the The webkit javascript console multiline editor and console.

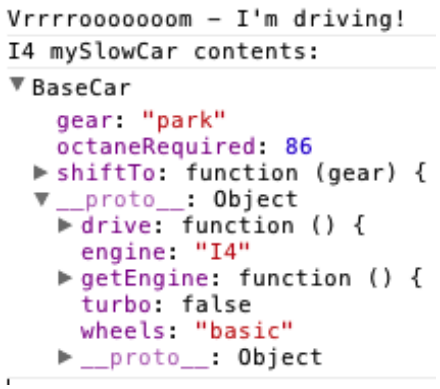

Figure 10.1 Instantiating an instance of `BaseCar` and exercising two of its methods

With our `BaseCar` class set, we can now focus on subclassing the `BaseCar` class. We'll first do it the traditional way. This will give you a better understanding of what's going on under the hood when we use `Ext.define` later on.

10.1.1 Inheritance with JavaScript

Creating a subclass using native JavaScript is achievable with multiple steps. Rather than simply describing them, we'll walk through the steps together. The following listing creates `PremiumCar`, a subclass of the `BaseCar` class.

Listing 10.2 Creating a subclass the old-school way

```
var PremiumCar = function() {                                          //1
    PremiumCar.superclass.constructor.call(this);                      //2
    this.octaneRequired = 93;
};
PremiumCar.prototype   = new BaseCar();                                //3
PremiumCar.superclass  = BaseCar.prototype;                            //4
```

```
PremiumCar.prototype.turbo  = true;
PremiumCar.prototype.wheels = 'premium';
PremiumCar.prototype.drive = function() {
   this.shiftTo('drive');
   PremiumCar.superclass.drive.call(this);
};
PremiumCar.prototype.getEngine = function() {
    return 'Turbo ' + this.engine;
};
```

#1 Configure subclass constructor
#2 Call superclass constructor
#3 Set subclass prototype
#4 Set subclass's superclass reference

To create a subclass, you begin by creating a new constructor, which is assigned to the reference `PremiumCar` #1. Within this constructor is a call to the `constructor` method of the `PremiumCar.superclass` within the scope of the instance of `PremiumCar` #2 being created (`this`).

You do this because, unlike other object-oriented languages, JavaScript subclasses don't natively call their superclass `constructor` #2. Calling the `superclass` constructor gives it a chance to execute and perform any `constructor`-specific functions that the subclass might need. In our case, the `shiftTo` method is being added and called in the `BaseCar constructor`. Not calling the superclass `constructor` would mean that our subclass wouldn't get the benefits provided by the base class `constructor`.

Next, you set the `prototype` of `PremiumCar` to the result of a new instance of `BaseCar` #3. Performing this step allows `PremiumCar.prototype` to inherit all of the properties and methods from `BaseCar`. This is known as *inheritance through prototyping* and is the most common and robust method of creating class hierarchies in JavaScript.

In the next line, you set the `PremiumCar`'s `superclass` reference to the `prototype` of the `BaseCar` class #3. You then can use this `superclass` reference to do things like create so-called extension methods, such as `PremiumCar.prototype.drive`. This method is known as an extension method because it *calls* the like-named method from the `superclass` prototype but from the scope of the instance of the subclass it's attached to.

> TIP All JavaScript functions (JavaScript 1.3 and later) have two methods that force the scope execution: `call` and `apply`. To learn more about `call` and `apply` visit the following URL: http://www.webreference.com/js/column26/apply.html.

With the subclass now created, you can test things out by instantiating an instance of `PremiumCar` with the following code entered into the The webkit javascript console editor:

```
var myFastCar = new PremiumCar();
myFastCar.drive();
console.log('myFastCar contents:');
console.dir(myFastCar);
```

Figure 10.2 shows what the output would look like in the The webkit javascript console multiline editor and console.

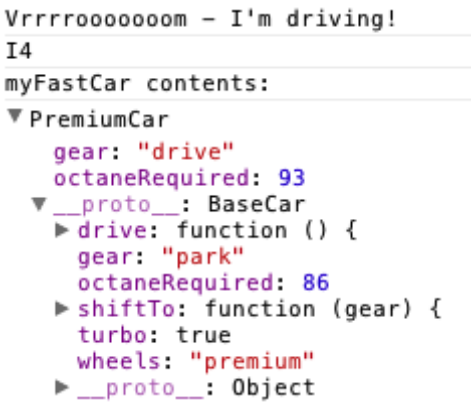

Figure 10.2 Our `PremiumCar` subclass in action

This output shows that our subclass performed as desired. From the `console.dir` output, you can see that the subclass constructor set the `octaneRequired` property to 93 and the `drive` extension method even set the `gear` method as `"drive"`.

This exercise shows that you're responsible for all of the crucial steps in order to achieve prototypal inheritance with native JavaScript. First, you had to create the constructor of the subclass. Then, you had to set the prototype of the subclass to a new instance of the base class. Next, for convenience, you set the subclass's `superclass` reference. Last, you added members to the prototype one by one.

You can see that quite a few steps need to be followed in order to create multiple classes with the native language constructs. I don't know about you, but I think this is a lot of work and utterly frustrates me. Luckily, there's an easier way to achieve this result.

Next, we'll show how the Sencha Touch class system makes creating classes and multiple inheritance much easier.

10.2 Using the Sencha Touch Class system

The Sencha Touch class system takes JavaScript's prototypal inheritance to another level bringing features like dependency injection, automatic setter and getter method creation, statics and mixin support (multiple inheritance). All of these features require the use of Sencha Touch class-specific methods, such as `Ext.define`, `Ext.create` and `Ext.require`, as you'll learn later on. As we walk though this section, you're going to learn why the Sencha Touch class system is a great solution for developing your applications.

10.2.1 Using Ext.define

We recently saw what it takes to implement JavaScript prototypal inheritance. We had to do quite a bit of work to just get a single level of inheritance setup. With complex software like

our applications, we'd have to do quite a bit of typing to get inheritance going. This means that we'd have a lot of redundant code in our projects, resulting in a lot of bloat.

Sencha Touch is in the perfect position to take on the heavy lifting for us and even adds a bit of automation. In order to see what I mean, we'll have to take a step back to the first two classes we created. We'll begin by redefining the BaseCar class we constructed just a bit ago. Along the way I'm going to introduce you to the "config system". Afterwards, we'll use extend the BaseCar class to define PremiumCar.

WE'LL BE USING NAMESPACES!

Instead of just using Simple class names, we'll be defining our class names using properly name spaced class names, much like you find in Classic languages like Java or C++. If you're new to namespaces, the only thing to them is that they help organize your code much like folders do on a file system. When developing applications you will need to organize your code according to namespace, so it's good to get used to it now.

Here's the BaseCar class defined in the MyApp.car namespace. We'll be extending Ext.Component, since that's where you'll be spending a lot of your time when developing applications.

Listing 10.3 Defining our base class with Sencha Touch

```
Ext.define('MyApp.car.BaseCar', {                                    // 1
    extend      : 'Ext.Component',                                   // 2
    requires    : 'Ext.Component',
    config      : {                                                  // 3
        octaneRequired : null,
        gear           : null,
        engine         : 'I4',
        turbo          : false,
        wheels         : 'basic'
    },
    constructor : function() {                                       // 4
        this.callParent();
        this.setOctaneRequired(86);
        this.setGear('park');
    },
    drive       : function() {
        console.log("Vrrrrooooooom - I'm driving!");
    }
});
```

#1 Define the BaseCar class
#2 Extend Ext.Component
#3 Define the config object properties
#4 The constructor

The listing above demonstrates the most basic use of `Ext.define`, where define MyApp.car.BaseCar **#1**. The first thing that might strike you as strange is the way you define the class names – by String.

This is because Sencha Touch gives you an opportunity to define a class and name space together. Sencha Touch will actually create the "MyApp.car" namespace for you if it is not previously defined and place the BaseCar class within that namespace. This pattern paves the way for us to look at using the class Loader system later on, where you'll learn how important it is to organize your classes in your project's file system according to namespace for which they are defined. But, let's continue with looking at this class. We have a lot to discuss.

In the class definition, we instruct the class system to extend Ext.Component**#2**, which is now known as the superclass for our BaseCar class.

Why extend Component?

We are extending Ext.Component because it gives us the capabilities to use the automatically generated setters and getters. Since most of your work will be done extending classes that have these magic functions created, it makes sense to start getting used to it now.

We also instruct Sencha Touch to require Ext.Component. This allows us to fully utilize the class loader dependency injection system, which is absolutely necessary to use the Sencha SDK, which we'll cover in Chapter 13.

Next, we define what is known as the class `config` object. This object is somewhat special as it is used to define custom properties will be set and accessed after instantiation. When you define classes, setter and getter methods are automatically generated for you. It also, gives you the capabilities to define custom properties in a single place, instead of the prototype of the class.

In the constructor, we immediately call the superclass's constructor. This allows the superclass to do what it needs to do in its own Constructor.

We can see this in action by instantiating an instance of our custom class. To instantiate this class, we'll have to use `Ext.create` instead of the JavaScript `new` keyword. By now, you should be used to using `Ext.create` for Sencha Touch classes, but I want you to use it within the context of your own class. Here's how you do it.

```
var mySlowCar = Ext.create('MyApp.car.BaseCar');
mySlowCar.setGear('drive')
mySlowCar.drive();
console.log(mySlowCar.getEngine());
```

As expected, here are the expected results.

```
Vrrrroooooooom - I'm driving!
I4
```

Figure 10.3 The results of our first Sencha Touch class.

As Figure 10.3 demonstrates, we have the expected results of our implementation of the `MyApp.car.BaseCar` class we just created.

Notice how we're using the `setGear()` and `getEngine()` methods. These are magic methods automatically generated by the class system. Right away, you should start to see how the Sencha Touch class system brings power to our code, by generating API methods to our custom class. There's more to the class system than what we just covered.

With our custom class instantiated, we can extend it using `Ext.define`. In doing so, I'm going to show you more of how the class system works.

Here's the code for the PremiumCar class, extending our BaseCar class.

Listing 10.4 Extending our BaseCar with Ext.define

```
Ext.define('MyApp.car.PremiumCar', {                                    // 1
    extend      : 'MyApp.car.BaseCar',                                  // 2
    config      : {                                                     // 3
        turbo  : true,
        wheels : 'premium',
        stereo : '5.1'
    },
    constructor : function() {                                          // 4
        this.callParent(arguments);
        this.setOctaneRequired(93);
    },
    applyEngine   : function(engine) {                                  // 5
        return 'Turbo ' + engine;
    },
    drive       : function() {                                          // 6
        this.setGear('drive');
        this.callParent();
        console.log('The turbo makes a big difference!');
    }
});
```

#1 Define the PremiumCar class
#2 Extend the BaseCar class
#3 Override the BaseCar config primitives
#4 Create the PremiumCar constructor
#5 Add an "apply" method
#6 Extend the drive method

In listing 10.3, you create the `MyApp.car.PremiumCar` class, which is an extension (subclass) of the `MyApp.car.BaseCar` superclass using the `Ext.define` method. Here's how this works.

We first call upon `Ext.define` you define the `MyApp.car.PremiumCar` **#1** extension class. We instruct Sencha Touch to extend our previously defined `MyApp.car.BaseCar` class by naming it via `String` representation set to the `extend` keyword **#2**.

Next, we set the config **#3** object to override the `turbo`, `wheels` and `stereo` properties, inherited from the superclass. Even though we override those three properties, the other config properties form the superclass will be inherited for us.

THE CLASS SYSTEM MANAGES INHERITANCE FOR CONFIG PROPERTIES

One of the things I want you to recognize and remember is that the class system will manage what is known as "deep merging of config properties". That is the extension class will inherit properties that are defined in the superclass. This is very much like basic prototypal inheritance, except it adds those automatically generated setters and getters for you.

When thinking about extending classes, you must consider whether prototypal methods in the subclass will share the same name as prototypal methods in the base class. If they will share the same symbolic reference name, you must consider whether they'll be extension methods or overrides.

An extension method is a method in a subclass that shares the same reference name as another method in a base class. What makes this an extension method is the fact that it includes the execution of the base class method within itself. The reason you'd want to extend a method would be to reduce code duplication, reusing the code in the base class method.

The constructor **#4** for this classis is an extension method. It's exact duplicate of the previously created `PremiumCar` constructor, with the addition of the call to the parent class constructor via `this.callParent(arguments);`. This statement allows the subclass to chain the constructor method calls, effectively allowing the `MyApp.car.BaseCar` superclass constructor to execute within the scope of new instances of our `MyApp.car.PremiumCar` subclass.

We are doing something a bit different in this class, adding what's known as an "apply" method (or applier function). An "apply" method is called by a setter method and gives your class an opportunity to do something with the value being set. In our example, we are prepending "Turbo " to the set engine type. So every time the auto-generated setEngine() method is called, the engine type will always have a turbo applied.

SENCHA TOUCH USES A UNIQUE CLASS INHERITANCE PATTERN

Internally, Sencha Touch uses the config system for every class and apply methods are in a lot of the classes. For the Component base class, apply methods are used to do things like construct instances of classes via a factory method known as Ext.Factory. The reason

apply methods exist is to abstract the construction of instances of classes from a setter method.

The drive method **#6**, is an extension method as well. In this method, we automatically set the gear to drive then call the parent method. This allows us to simply call drive on instances of this extension class, and it automatically set the gear for us.

YOU CAN ALSO OVERRIDE METHODS

Even though we are not using it, it's good to discuss what an override is. An override is a method in a subclass that shares the same name as another method in a super class but doesn't chain method calls up to the superclass via `this.callParent()`. You override a method if you wish to completely discard the code that's in the like-named method in the base class. Therefore, you just don't call this.callParent() within the method, and the upwards execution chain will not occur.

Now that you have your `PremiumCar` configured using `Ext.define`, you can see it in action using the webkit javascript console. You can do so using the exact same code you used when exercising your manually created subclass:

```
var myFastCar = Ext.create('MyApp.car.PremiumCar');
myFastCar.drive();
console.log(myFastCar.getEngine());
```

Figure 10.4 shows what it looks like in the The webkit javascript console console.

```
Vrrrrooooooom - I'm driving!
The turbo makes a big difference!
Turbo I4
```

Figure 10.4 The results of the instantiation of the `PremiumCar` class

You've just successfully extended a class using `Ext.define`. What I've shown you are just the very basics of using `Ext.define`, where we created a class from scratch (`MyApp.car.BaseCar`) and extended it (`MyApp.car.PremiumCar`).

From here, we can extend the `MyApp.car.PremiumCar` class and create a `MyApp.car.SportsCar` class that adds features like the ability to do fun things like `drift`()-ing and perhaps `dragRace`()-ing;

All of what we've done with prototypal inheritance with JavaScript and `Ext.define` then positions us to start exercising our newly found knowledge and start extending Sencha Touch.

10.3 Extending Sencha Touch Components

Extensions to the framework are developed to introduce additional functionality to existing classes in the name of reusability. The concept of reusability drives the framework, and when utilized properly it can enhance the development of your applications.

Some developers create preconfigured classes, which are constructed mainly as a means of reducing the amount of application-level code by stuffing configuration parameters into the class definition itself. Having such extensions alleviates the application-level code from having to manage much of the configuration, requiring only the simple instantiation of such classes. This design pattern is OK, but should only be used if you're expecting to stamp out more than one instance of this class. Else, it's considered wasteful just to define a Class just as a home for a collection of configuration parameters.

Other extensions add features such as utility methods or embed behavioral logic inside the class itself. An example of this would be a `form.Panel` that automatically pops up a `MessageBox` whenever a save operation failure occurs. I often create extensions for applications for this very reason, where the widget contains limited built-in behavioral logic. I say limited, because with Sencha Touch 2.0, we now have an MVC architecture for which we can abstract business logic to controllers. This is something that we'll look into in the next chapter.

We're going to develop an extension to Sencha Touch's List widget and then look at some pros and cons for extending a widget versus creating a plugin. This is going to be fun!

10.3.1 Thinking about what we're building

Many phone and tablet applications use Lists of some sort. Often times to invoke an action (such as delete or update), you tap on a list item, and another screen appears allowing you to invoke that action. What if I told you that instead of actually having that action screen appear, you could simply display the actions in the list item itself?

It turns out that with a little bit of work, you can extend the List class to allow you to inject this little bit of functionality, making your applications much more intuitive and easier to use.

To make the picture clearer, here's an illustration of our extension flipping a list item.

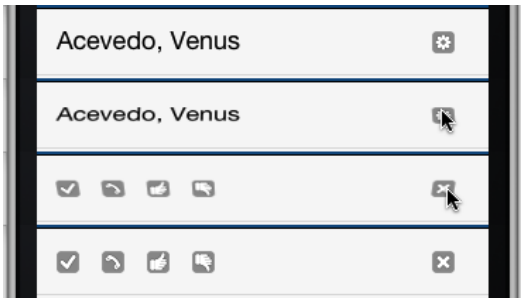

Figure 10.5 Four frames demonstrating our first extension in action, where we flip a row in 3D space.

As figure 10.5 depicts, tapping on the gear icon will flip the row, revealing icons for the user to tap. When the user taps on an action, we'll be firing a custom event.

Let's get to work.

10.3.2 Extending the List dataview

This extension is just shy of 140 lines. And it is pretty complex from an implementation perspective. In order to display the row, we're going to have to do things like dynamically inject DOM elements with the icons, flip the row and when an action is clicked or someone scrolls, flip the row back and destroy the dynamically injected element.

> **Don't forget the CSS**
>
> In order to get this extension to work properly, we'll have to use two the CSSs files I generated. They will be in the zip file you downloaded under examples/chapter10/list_actions.css and chapter10/icons/icons.css. Be sure to include these in your project.

We'll have different methods to do all of this work, but I want to be able to describe to you what they do individually.

Because the extension is so large, we're going to code in chunks, beginning with a skeleton of the class.

Listing 10.5 Our ActionList extension Template

```
Ext.define('Ext.ux.ActionList', {                                    // 1
    extend : 'Ext.dataview.List',
    xtype  : 'listactions',                                          // 2
    config : {                                                       // 3
        actionsEl    : null,
        actions      : null,
        itemTemplate : undefined                                     // 4
    },
    initialize : function() {},

    applyItemTemplate : function() {},                               // 5

    onItemTapDoFlip : function(view, idx, itemEl, record, evtObj) {},

    clearActionsEl : function() {},

    destroy : function() {}
});
```

#1 Define our class
#2 Configure the xtype
#3 Setup the config object
#4 Configure a template stub configuration
#5 The itemTemplate applier function

The above template has quite a bit going on and definitely requires some explaining. It begins with the definition of our User Extension (ux) class, ActionList**#1**, which extends Ext.List. In order to allow for lazy instantiation, we configure our class with the xtype property**#2**, listactions. This will allow us to configure our extension as a child to any other Container, just like any other framework widget.

NAMESPACE FOR YOUR PLUGINS

We place our extension in the Ext.ux namespace#3. Since this is a freely available plugin that I developed, it makes sense to place it there. If you were developing plugins for your own company or application, you might want to place it in your own plugins or similar namespace.

Next, we setup the `config` object**#3**. The `actionsEl` property will be used as a reference to the to rendered DOM with the icons for the user to tap. `actions` is a user-configured set of objects that will define what icons will be rendered on each list item.

What icons?

The list of icons is contained in the icons.css file I described just a short bit ago. I generated it via a small BASH (UNIX shell) script that I generated. It's easy to read and implements the icons that were included in the package and are free to use per the license described in the Readme.txt file inside of the examples/chapter10/icons directory.

The `itemTemplate` property**#4** will be used as a local reference pointing to an instance of Ext.XTemplate that we'll be using to render the icons on each list item. You'll see how this all works when we talk about the `applyItemTemplate` *applier* function **#4.**

After the config object, we have our methods, starting with initialize. In the initialize method, we will be doing things like registering event handlers. The onItemTapDoFlip method is an event listener for the item tap, and is responsible for applying the respective CSS3 animations to flip the item over, revealing the action icons. clearActionsEl is a method that is charged with cleaning up the DOM when an action is chosen and the list item flips back to its original state (showing the list item data). Lastly, destroy is a method responsible for clearing up event listeners and ensuring that the DOM created by this plugin is completely purged, paving the way for garbage collection to alleviate memory burden.

With our class template now defined, it's time to start filling in these methods. We'll begin with `initialize` and `applyItemTemplate`.

Listing 10.6 ActionList initialize and applyItemTemplate methods

```
initialize : function() {
    var me = this;                                                    // 1

    me.getItemTpl()
```

```
        .html += '<span class="target-add list-icon gear"></span>';     // 2

    me.on({                                                             // 3
        scope   : me,
        itemtap : 'onItemTapDoFlip'
    });

    me.getScrollable().getScroller().on({                               // 4
        scope  : me,
        scroll : 'clearActionsEl'
    });
    me.callParent();                                                    // 5
},

applyItemTemplate : function(cfg) {
    if (! cfg) {
        return Ext.create('Ext.XTemplate',                              // 6
            '<div class="flexbox">',
                '<tpl for=".">',
                    '<div class="list-action list-icon {action}"> </div>',
                '</tpl>',
                '<span class="target-remove list-icon cancel"> </span>',
            '</div>'
        );
    }
},
```

#1 Minification-friendly reference
#2 append to the configured passed List itemTpl
#3 Register item tap event
#4 Register the scroller's scroll event
#5 Call the superclass's initialize method
#6 Create an instance of Ext.XTemplate

The first method we're going to explore is initialize**#1** and is responsible for a few things. First, we set a minification-friendly lexically scoped reference, me, to the magic reference of this. This pattern reduces the over-the-wire burden on our web devices by being transformed to a single letter during the production build phase, which we'll cover next chapter.

MINI-WHAT?

Minification is a technique used by man JavaScript developers to reduce the over all size of your JavaScript files to allow for faster over-the-wire transfer and speed up interpretation of your code. However, it is generally only deployed in production environments as it is extremely difficult to debug! There are a few phases to minification. First is the concatenation of files into a single one. Next is the removal of comments and all white space. The last is more aggressive and yields the best results, which is called obfuscation. This is where your JavaScript is transformed (obfuscated) to reduce a higher yield in code reduction. We'll look at how we can use the Sencha SDK tools to minify your code in Chapter 13.

This method also is responsible for key functionality, such as injecting a String representing an HTML fragment **#2** for the list items, which will render as the gear icon. It also registers a tap event handler on the component's element**#3** and a scroll handler on the Component's scroller**#4**. We'll dive into what these event handlers do a little later on. The last thing the initialize method accomplishes is calling the chaining the superclass' initialize method call via the local callParent method**#5**.

The applyItemTemplate is a typical factory method and is automatically called by the class's config system because Component's constructor specifically calls on initConfig, which is responsible for applying all configuration properties to the instance of the class.

applyItemTemplate is responsible for creating an instance of Ext.XTemplate for us. This XTemplate is responsible for writing the icons to the DOM as well as CSS class names that we'll be using to hook on to. One thing to pay particular attention to is the {action} token. This will be filled in via the data that is provided by the implementation.

THE APPLIER PATTERN IS EXTREMLY POWERFUL

Following this pattern allows us to accomplish a few things. First, if the itemTemplate property is not defined, it acts as default and gives us the future capability of customizing the instance of XTemplate based on other class parameters. Secondly, if you wanted to override the itemTemplate, you could configure an instance of this class with the instance of XTemplate and the if condition would not enter its true state, therefore your configured itemTemplate would act as an override to that particular instance.

Before we can see this extension in action, we'll need to finish up this class. We have quite a bit of work to do. Next, we're going to tackle the biggest method in our class, onItemTapDoFlip, which is responsible for quite a bit of work. This is the biggest method in our class, and I'm going to explain how it works in great detail.

Listing 10.7 ActionList onItemTapDoFlip event listener method

```
onItemTapDoFlip : function(view, idx, itemEl, record, evtObj) {
    var me              = this,                                    // 1
        isTargetAdd     = evtObj.getTarget('.target-add'),
        actionIcon      = evtObj.getTarget('.list-action'),
        ROTATE_RESET    = 'rotate-list-item-reset',
        ROTATE_BACKWARD = 'rotate-list-item-backward',
        ROTATE_FORWARD  = 'rotate-list-item-forward',
        actionsEl       = this.getActionsEl(),
        actionName,
        classList,
        eventName,
        animEl,
        parentNode;


    if (actionsEl) {                                               // 2
        me.clearActionsEl();
```

```
        }

        if (actionIcon) {
                                                                        // 3
            classList  = actionIcon.classList;
            actionName = classList[classList.length - 1];
            view.fireEvent(eventName, view, idx, itemEl,
                                        record, actionName, evtObj);
            Ext.defer(view.deselectAll, 1, view);                       // 4
            return;
        }

        if (isTargetAdd) {                                              // 5

            animEl = itemEl.down('.x-list-item-label');
            parentNode = Ext.get(animEl.dom.parentNode);

            actionsEl = Ext.Element.create({
                cls   : 'tmp-list-item x-list-item-label ' + ROTATE_FORWARD,
                html  : me.getItemTemplate().apply(this.getActions())
            });

            me.setActionsEl(actionsEl);
            parentNode.insertFirst(actionsEl);

            animEl.addCls(ROTATE_BACKWARD);
            animEl.removeCls(ROTATE_RESET);

            Ext.defer(function() {
                view.deselectAll();
                actionsEl.removeCls(ROTATE_FORWARD);
                actionsEl.addCls(ROTATE_RESET);
            }, 1);
        }
    },
```

#1 Setup lexically scoped references
#2 Purge existing actions DOM
#3 Fire custom event if an action icon is tapped
#4 Defer execution of the list item deselection
#5 Flip the list item & paint action icons

After reading the code for onItemTapDoFlip, I am sure you're aware that there is plenty of stuff going on here. Here's how this all works.

First, we create a bunch of lexically scoped references**#1**, some for reusability, but others for readability & ease of debugging. For example, the "is" references are the result of a target element acquisition attempts via the Ext.EventObject getTarget method calls. We use these to determine if the user tapped on the icon to show ('.target-add' CSS class) or hide ('.target-remove' CSS class) the action icons in the DOM. Other references include the actual action icon tapped (actionIcon) or the entire DOM element that contains the entire collection of icons (actionsEl).

Next, we begin a series of conditional statements, each testing for different interaction models or assertions. This is where most of the work happens in this method.

The first conditional test is to see if we have an existing DOM element that has icons rendered on screen (actionsEl)**#2**. If so, we call clearActionsEl(), which is responsible for removing previously rendered DOM. We do this first because we know that any level of interaction from the user with the list should hide the actions.

The next conditional tests if someone actually tapped on an action icon (not the gear)**#3**. If so, we interrogate the element's class list to extract the actual icon class name (remember the {action} token in the itemTemplate applier function?). The className is used to fire a custom event name like 'action_thumb_down', passing all possible data to the implementer, such as the reference to the view, the index of the list item tapped and so on. Lastly, we halt execution of this function via the return keyword.

WHY FIRE CUSTOM EVENTS?

We fire custom events because it offers the best level of flexibility for our extension. This means that when we go to build an application with MVC, we could, in theory, have multiple controllers listen to events from this view.

The last part of this if block is the de-selection of any prior selected ListItem, via a deferral of the view's `deselectAll` by one millisecond**#4**. This allows the Sencha Touch event system to finish what it needs to do and our JavaScript code to continue to execute, but shortly there after clear the selection.

WHY DEFER EXECUTION OF VIEW.DESELECTALL?

This is one of those "order of operations" must haves that is difficult to explain in the context of this chapter without into going into JavaScript timers and an extreme deep dive of the data view selection model internals. The important thing to know is that Sencha Touch itself has timeouts to deal with touch start, touch end and tap events, and the only way to guarantee the successful de-selection of a list item is to actually defer execution of deselectAll by one millisecond, which ensures that Sencha Touch's code comes to full completion, and allows our code to execute almost immediately after.

The biggest block of code test to see if someone tapped the gear icon**#5**, which was appended to each list item inside of the initialize function. This block first gathers references for the element to be animated (list item), its parent node and creates a new DOM element via Ext.element.create. This new DOM element (actionsEl) contains all of the icons to be painted on screen and flipped in.

Then, we cache the actions element via the automatically generated setActionsEl method, and then inject it into the DOM via the parentNode.insertFirst method. At this point, the DOM element containing the icons and the existing list item are siblings in the

DOM tree. In order to make the DOM flip, we have to instruct the list item element to rotate backward via the addCls method, injecting the proper CSS3 animation class as defined in the list_actions.css. We immediately remove any pre-defined rotation reset declaration, allowing the animation to act smoothly. In order to have the actions element flip in at the right time, we have to defer its appearance by 1 ms.

I know! This seems like a hack. But, if I flipped both items at the same time, they collide, as illustrated below.

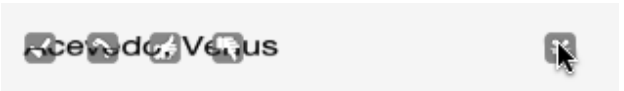

Figure 10.6 If you incorrectly time the rotation of items in 3D space, they can bleed on top of each other, making the animation less pleasing to the eye.

The method we just covered seems like an absolute monster on paper, but in reality it's only a tad over 50 lines.

OK. We're almost done. I have some relief coming your way! Next, we'll be covering the clearActions and destroy methods. These are smaller.

Listing 10.8 ActionList clearActionsEl and destroy methods

```
clearActionsEl : function() {
    var tempEl  = this.getActionsEl(),                          // 1
        ROTATE_RESET,
        parentNode,
        animEl;

    if (tempEl) {                                               // 2
        ROTATE_RESET = 'rotate-list-item-reset';
        parentNode   = tempEl.parent('.x-list-item');

        animEl = parentNode.query('.x-list-item-label').pop();
        animEl = Ext.get(animEl);

        animEl.addCls(ROTATE_RESET);                            // 3
        animEl.removeCls('rotate-list-item-backward');

        tempEl.addCls('rotate-list-item-forward');              // 4
        tempEl.removeCls(ROTATE_RESET);

        this.setActionsEl(null);                                // 5

        Ext.defer(function() {                                  // 6
            tempEl.destroy();
            Ext.removeNode(tempEl);
        }, 300)
    }
},
destroy : function() {
```

```
        var tempEl = this.getActionsEl();                          // 7
        if (tempEl) {
            this.setActionsEl(null);
            tempEl.destroy();
            Ext.removeNode(tempEl);
        }
        this.callParent();
    }
```

#1 Gather references
#2 Perform actions only if actions element exists
#3 Rotate list item back in
#4 Rotate list actions element out
#5 Clear local actionsEl member
#6 Destroy actions element after animation
#7 Purge actions element upon component destruction

The clearActions method is responsible for animating out existing action icons from the screen and is the second most complex method we have compared to the onItemTapDoFlip method we covered a short bit ago. Here's how it works.

The first thing that happens is the setting of the tempEl reference **#1** via a call to this.getActions(). That reference is then tested to be truthy, and if so, we get on with the rest of the work.

To perform the proper animations, we must first get a reference to the elements to animate; the list item (animEl) and the actions element (tempEl) **#2**. Those references are used to rotate the list item into view **#3** and the actions out of view **#4.** After the animations start, we set the local actionsEl member to null and then defer the execution of the destruction of the DOM that represents the action icons by 300 milliseconds. Deferring the execution of this code block ensures that the animation completes before the DOM for the action icons are removed.

The last method in our class, destroy, is simply responsible for removing any existing actions element from the DOM, but does so immediately. This method comes in handy when a list rendering the actions is destroyed and ensures proper cleanup.

Our extension is now complete. We can now look at implementing this beast.

10.3.3 Implementing our extension

In order to implement our extension, we'll need to create an instance of Ext.List. Here's what a simple implementation would look like with four action icons and a generic event handler for all of all event types.

Here's the implementation of our extension.

Listing 10.9 ActionList extension in action

```
Ext.define('ListModel', {                                          // 1
    extend : 'Ext.data.Model',
    config : {
        fields : [
            'lastName',
```

```
            'firstName'
        ]
    }
});

var store = Ext.create('Ext.data.Store', {                          // 2
    model    : 'ListModel',
    autoLoad : true,
    proxy    : {
        type : 'ajax',
        url  : 'complexData.json'
    }
});

var handleAction = function(view, idx, itemEl,
          record, actionName, evtObj) {                             // 3
    Ext.Msg.alert(
       'You tapped ' + actionName,
       record.get('firstName') + ' '
           + record.get('lastName')
    );
};

Ext.create('Ext.ux.ActionList', {                                   // 4
    fullscreen : true,
    store      : store,
    itemTpl    : '{lastName}, {firstName}',
    items      : [
        {
            xtype  : 'toolbar',
            docked : 'top',
            title  : 'Actions List extension!'
        }
    ],
    actions : [                                                     // 5
        'ok',
        'phone',
        'thumbs_up',
        'thumbs_down'
    ],
    listeners : {                                                   // 6
        action_ok         : handleAction,
        action_phone      : handleAction,
        action_thumb_up   : handleAction,
        action_thumb_down : handleAction
    }
});
```

#1 Define the model
#2 Create the store
#3 A generic action event handler
#4 Create an instance of our extension
#5 Configure our set of actions
#6 Setup the event listeners

To implement our extension, we need to do a little bit of work, beginning with the definition of the model**#1** and its required data Store **#2**. Next, we create a generic event handler for all of the custom events**#3**. This generic handler is simply going to show a MessageBox alert displaying what action you tapped and the first and last name of the record you tapped, which demonstrates custom the custom event data being fired.
Lastly, we create an instance of our ActionList extension**#4**, where we configure the list of actions**#5** and their related event listener registrations #6.

That's pretty much all there is to implementing this extension. Here's what it looks like in action.

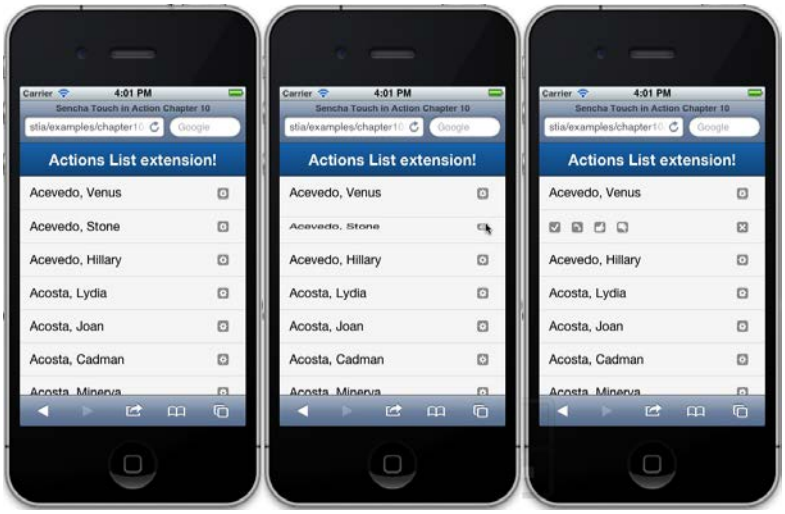

Figure 10.7 Our Actions List extension will render the gear icon to the right of the text (left) and when a user taps on the gear (center), it will animate that row revealing the actions (right).

We can see that our extension is properly rendering the gear icon to the right of the itemTpl text and flips the row, per Figure 10.7. But, to test the events, we'll need to tap on an action as illustrated below.

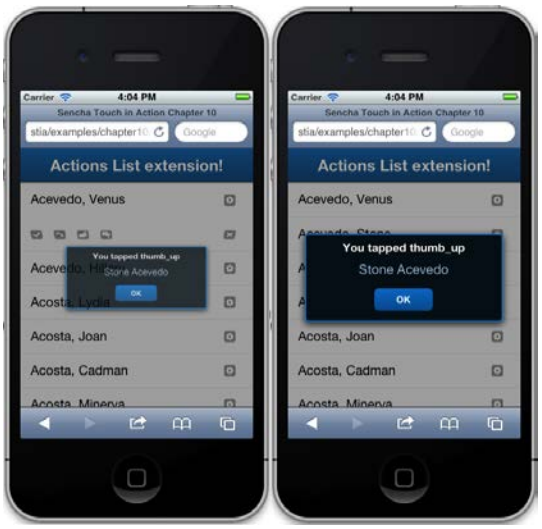

Figure 10.8 exercising the events fired by our extension.

Figure 10.8 above illustrates how when a user taps on an action, the row flips and the MessageBox alert appears, echo'ing out the row you tapped and the action icon that was tapped.

This extension could be used to allow users to do a lot of things to records in your application, much like call someone, vote up or down a record or delete it. It can be customized to invoke custom icons (a-la changing the CSS file) to give it a much more polished or custom look for your application.

We just concluded a pretty awesome exercise, where we extended a List data view, and injected custom DOM to give unique controls to our apps that is not available from the framework out of the box.

10.3.4 Thinking about inheritance limitations

But, there are some major limitation to extending components, and is something that you need to consider. Say for example, you are working with a team that are all developing the same application with you and a requirement comes up where you have to decorate three lists in your application that already exists, each with different basic business rules. How do you accomplish this with an extension?

Perhaps you could create a base class that all of your classes inherit from that injects functionality. At this point in your application's development cycle that induces risk, which is something that I strongly caution against, unless your project sponsors are OK with this risk. But chances are, they are not due to the tight deadlines that all of us love to be on the hook to fulfill.

So what alternatives to do we have? The only real solution to this problem is a plug-in.

10.4 Plug-ins to the rescue

Plugins solve this exact problem by allowing developers to distribute functionality across widgets and classes without having to deal with multiple inheritance. What also makes plug-ins powerful is the fact that you can have any number of them attached to an instance of `Component`.

> **COMPONENT LIFECYCLE REFRESH**
>
> In case you don't remember when exactly Plugins are created and initialized, now would be an excellent time to brush up on the initialization phase of the component lifecycle, which we covered in Chapter 3.

Before we dive right into creating plugins, we should have a quick chat about how plugins work.

10.4.1 The anatomy of a plugin

The basic anatomy of a plug-in is simple: it starts out by defining your class. If you want to extend an existing Sencha Touch class, it's OK to do so. For our plugin, we'll be extending Component, as we'll need to inject DOM into our List.

```
Ext.define('MyApp.plugins.MyPlugin', {
    extend : 'Ext.Component,
    alias  : 'plugins.myplugin',
    // class related stuff
});
```

The above snippet demonstrates the very basics of creating a plugin using `Ext.define`, where we setup alias as 'plugins.myplugin'. We prefix our plugin alias with 'plugins' to allow the Sencha Touch class management system to route the registration of this class with `PluginManager`, which will be responsible for creating instances of our classes via lazy objects, much like XTypes, except they are known as Plugin Types in this case.

Here's an example of how we'd use a lazy object to configure this plugin in a generic Component instance:

```
Ext.create('Ext.Component', {
    plugins : [
        {
            type : 'myplugin'
        }
    ]
});
```

In the above code, we create an instance of `Ext.Component` and set its plugins property to an array with a single object. The single object has a `type` property set as 'myplugin'. When the component nears the end of its initialization phase, it will create an instance of our custom plugin via this `type` shortcut.

The above pattern is considered the best practice by many in the industry, and uses the Sencha Touch lazy instantiation mechanism. There is an alternative pattern that uses the fully qualified class name and looks like this:

```
Ext.create('Ext.Component', {
    plugins : [
        {
            xclass : 'MyApp.plugin.MyPlugin'
        }
    ]
});
```

The difference between the two patterns is the first one (using type), requires the class itself to be defined in memory. If you used the xclass pattern and that class is not in memory, Sencha Touch will require it on demand via the loader system, which could cause a performance blip in your app while the network request is going on for the required resource. The best way to avoid this is to define your plugins as required classes in your application. In case you're wondering, we'll be covering this in Chapter 11.

The theory of plugins is rather simple but in order to fully understand how this stuff works we'll have to actually put it in practice.

10.4.2 Developing our plug-in

Remember that we're converting our prior List extension to a plugin, which will mean that we'll be seeing some code that we've already gone over. For completion, we'll do a quick review of the class, but I will leave out the redundant code. I'll make sure to cover the differences in great detail so that you understand how this differs from the extension we just made.

Here's our plugin, called ListActions. When reviewing this, please remember to look at the implementation code from your download.

Listing 10.10 Our custom ListActions plugin

```
Ext.define('Ext.ux.ListActions', {                                    // 1
    extend : 'Ext.Component',
    alias  : 'plugin.listactions',                                    // 2
    config : {
         // same config as our extension
    },

    init : function(parent) {
        var me = this;
        me.parent = parent;                                           // 3

        parent.getItemTpl().html += '<span class="target-add list-icon
gear"></span>';

        parent.on({
            scope   : me,
            itemtap : 'onItemTap',
            destroy : 'destroy'
        });
```

```
            parent.getScrollable().getScroller().on({
                scope  : me,
                scroll : 'clearActionsEl'
            });
        },

        applyItemTemplate : function(cfg) {
            // same code as our extension
        },

        onItemTap : function(view, idx, itemEl, record, evtObj) {
            // Same code as our extension
        },

        clearActionsEl : function() {
           // same code as our extension
        },
        destroy : function() {
           // same code as our extension.
        }
});
```

#1 Define our ViewContextMenu plugin
#2 Set the plugin type alias
#3 Define the init method

Our plugin code begins with the definition of the class via the namespace Ext.ux.ListActions**#1**, which extends Ext.Component. We extend Component because it gives us automatic config system integration. Immediately after the class definition, we set its alias to "plugin.listactions" **#2**. This will allow us to easily configure it in a List's plugins configuration option via lazy instantiation.

The next thing we should focus on is the init method **#3**. If you can recall the initialize method of our extension, the init method should seem extremely similar. In fact, it's 99% the same. The difference between the extension and this init method is the registration of me.parent set to the passed parent. A plugin is initialized via its init method. Its parent component is responsible for calling that method, and passes a reference to itself as the first argument. Inside of the plugin, we set the me.parent local member to the passed argument for future reference.

A QUICK WORD ABOUT THE EVENTS FIRING FROM OUR PLUGIN

Recall that our ActionList extension fired custom events based on which action icon was tapped. We'll be doing the same thing from our plugin (via onItemTap). The reason we're firing the event from the parent List instead of the plugin is because if we were to embed this in an application, it would be much easier to latch on to a view's event from a Controller than a view's plugin.

From there, the rest of the init method is nearly exactly the same. We register the item tap and destroy event handlers, as well as the parent List's Scroller scroll event. All of the other methods are exactly the same as our extension, so we don't need to cover them.

This leads us to an instantiation of this plugin, which again will look very familiar. To do this, we'll simply copy and modify listing 10.XX to save some time. The following listing will only contain the List instance and our plugin configuration.

Listing 10.11 ListActoin plugin in action

```
Ext.create('Ext.dataview.List', {                                    // 1
    fullscreen : true,
    store      : store,
    itemTpl    : '{lastName}, {firstName}',
    plugins    : {
        xclass  : 'Ext.ux.ListActions',                              // 2
        actions : [                                                  // 3
            'ok',
            'phone',
            'thumb_up',
            'thumb_down'
        ]
    },
    items      : {
        xtype  : 'toolbar',
        docked : 'top',
        title  : 'ListAction plugin'
    },
    listeners  : {
        action_ok         : handleAction,
        action_phone      : handleAction,
        action_thumb_up   : handleAction,
        action_thumb_down : handleAction
    }
});
```

#1 Configure a List dataview instance
#2 Setup our ListActions plugin via lazy instantiation
#3 Configure the actions

To implement our plugin, we have to configure a List **#1**. Inside of that list, we set the plugins attribute to an object, with an xclass property of Ext.ux.ListActions**#2**. This object will be used by the List instance to be transformed into an instance of our plugin, which is why we set the actions property**#3** on it.

Here's the plugin in action:

Figure 10.8 our ListAction plugin working inside of an instance of List.

As we can see in Figure 10.8, the list action plugin works perfectly inside of the List instance. We tap on the gear icon, and the action row is revealed, allowing us to tap on an action. As predicted, here's what happen when you tap on an action.

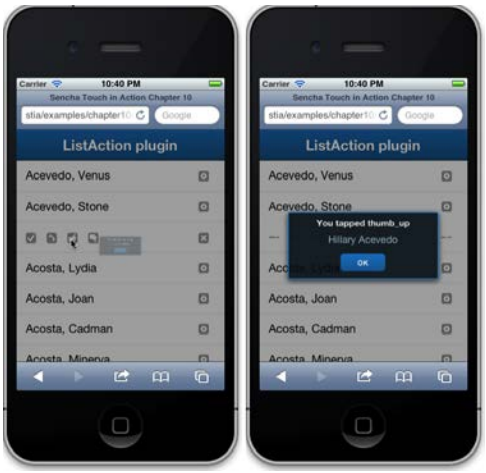

Figure 10.9 The results of tapping on an action via our ListAction plugin.

Figure 10.9 illustrates the predictable behavior of a message box being rendered on screen revealing which action was tapped and what the first and last name is for the record that was

tapped.

There you have it! A plugin we developed using our prior extension as a template. Remember that plugins are used to inject features into a component and if they inject DOM into the browser, you must be careful to properly clean things up when the parent component is destroyed.

This concludes the exploration of plug-ins. By now, you have the necessary basics to create plug-ins that can enhance functionality in your projects. If you have an idea for a plug-in and aren't too sure if it's been done before, visit us at the Sencha Touch forums via http://sencha.com/forum. An entire section is dedicated to user extensions and plug-ins, where fellow community members have posted their work, some of which is completely free to use.

Sencha has a nice collection of user extensions that you can download at its own market place. You can visit it at http://market.sencha.com.

We now understand the mechanics of developing an extension to the framework and also constructing a plugin using actual real-world requirements.

10.5 Summary

In this chapter, you learned how to implement the prototypal inheritance model using the basic JavaScript tools. During that exercise you got to see how this inheritance model is constructed step by step. Using that foundational knowledge, we refactored our classes using the `Ext.define` class definition method.

Next, we took all of our foundational knowledge and applied it to the extension of a List, where we were able to inject custom functionality to flip a row and reveal custom DOM that was injected by our extension. We also took the opportunity to fire custom events based on the configuration that was passed to our List extension.

Lastly, we took a look at some of the inheritance limitations extensions can pose and how developing plugins can resolve these limitations. From there, we converted our extension to a plugin and looked at how we can implement it.

In the next chapter, we're going to look at how you actually work with the SDK tools to develop an application for deployment. Afterwards, we'll take a deep dive into an application that I developed for Sencha called "Discover Music", where you'll begin to see a lot of the "ins and outs" of developing a complex application with Sencha Touch.