In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 2 'MVC and MVVM'
- A synopsis of the book’s content
- More information on Ext JS Application Development Blueprints

About the Author

Colin Ramsay is a software developer and writer with 15 years of coding experience. From .NET to Ruby, JavaScript, and CSS, he has worked with a range of technologies and local and international clients. His company, Go Tripod Ltd., is based in Cornwall, UK. This company works with multinational clients to implement exciting JavaScript products and ideas. He has coauthored Learning Ext JS and Learning Ext JS 3.2, both by Packt Publishing. Colin also writes on various topics on his personal blog at http://colinramsay.co.uk/.

His main hobby is to make mischief.
Ext JS Application Development Blueprints

In times gone by, JavaScript was used to provide minor enhancements to the Web in order to decorate pages with amusing effects or to validate user input. Slowly but surely, as browsers implemented faster JavaScript engines and new features and as developers and engineers came up with new techniques and ideas, JavaScript became near-ubiquitous in providing the functionality that end users have come to expect. With Ext JS, you have a toolset that allows you to build advanced JavaScript applications, but with great power comes great responsibility.

An application architect is the guiding hand of a project. They help to ensure that client expectations are met, timescales are achieved, and costs are contained. While juggling these requirements, the architect must provide the foundations for a technically successful project consisting of high quality, understandable code.

In Ext JS Application Development Blueprints, we'll go through the theory and practice of designing and building Ext JS applications. Over the next 11 chapters, you'll learn how Ext JS gives developers the key to a professionally designed software project and how to take this knowledge and use it in a range of real-world scenarios.

What This Book Covers

Chapter 1, Introduction, describes why software architecture is important and how this book will help you learn it.

Chapter 2, MVC and MVVM, introduces two classic patterns in software development and how they're implemented in Ext JS.

Chapter 3, Application Structure, shows how the various parts of an Ext JS application can be organized into a form that eases the development process.

Chapter 4, Sencha Cmd, looks at the way Sencha's command-line tools act as a powerful companion for your developers, enabling a leaner product that is delivered on schedule.

Chapter 5, Practical – a CMS Application, shows how to design and build a simple CMS application and how to start employing the ideas you've learned so far.

Chapter 6, Practical – Monitoring Dashboard, involves the creation of a log viewer dashboard using charts and grids that adapt to user input.

Chapter 7, Practical – an E-mail Client, shows off the responsive features of Ext JS 5 while continuing to demonstrate advanced use of the MVVM pattern.
Chapter 8, Practical – Questionnaire Component, uses Sencha Cmd to demonstrate code reuse by designing and building a wizard component.

Chapter 9, A Shopping Application, brings all the techniques from previous chapters to bear on a tablet-centric shopping app.

Chapter 10, Debugging and Performance, teaches you advanced techniques to diagnose a range of problems in development and production.

Chapter 11, Application Testing, provides you with techniques to build automated tests that allow for the kind of robust application expected from an application architect.
The problem with software development is that we're always looking for the right way to do things. Each software shop will have their own set of guidelines that indicate how their developers should operate. This is the way that software works: we build a set of ideas that reflect our best thoughts on how things should be developed and the software community learns from these ideas and builds on them. They are formalized into patterns of working and these patterns are shared throughout the development community. In this chapter, we'll talk more about this concept, specifically:

- The MVC pattern
- The MVVM pattern
- The way Ext JS uses both
- The evolution of Ext JS from MVC to MVVM
- The benefits of design patterns in the current version of Ext JS

Discussions about design patterns are often very dry. In this chapter, we'll use some practical examples to illustrate why they're so important and how they can help you kick-start your architecture efforts.

Diary of always

In the beginning, there was a giant mess. Well, maybe not quite, but in modern software development, we've got lots of design and architectural patterns that we can draw on to help us shape an application and ensure we're not reinventing the wheel. Each of these is the result of decades of their work, which is constantly reviewed and put into practice, and we all hope that the most elegant and useful work will bubble to the top. Along the way, we've seen clumsy patterns being overtaken by more elegant ones. Hopefully, our mess has become a little bit less tangled.
A key development in the way we build graphical interfaces was **model-view-controller (MVC)**, which was invented at the near-legendary Xerox PARC in the 1970s by Norwegian computer scientist Trygve Reenskaug. It was first publicly incorporated in Smalltalk, a programming language developed by a cast of computer scientists including Alan Kay. It brought together a host of ideas, which influenced nearly all the object-oriented languages we use today. It was a pretty big deal and created by some pretty big guns.

Connelly Barnes, assistant professor of computer science at University of Virginia, gives us a great way of viewing MVC:

"The model is the data, the view is the window on the screen, and the controller is the glue between the two."

It was first in describing software constructs in terms of their responsibilities, for example, the View is responsible for presentation. In **Chapter 1, Introduction**, we talked about its importance in creating strong application architecture.

It can be difficult for us to look back on innovations (such as Smalltalk and MVC) and understand why they were so important. We could spend many pages reviewing what went before and why the advent of MVC has been described as a seminal insight. What really matters though is that it was a new way of looking at organizing graphical user interface-based software, a new paradigm in computing science that would prove to stand the test of time for the next thirty years:

![Martin Fowler's bare-bones MVC](image)

There are several differences between the MVC implementation that Ext JS uses (the one that Ruby on Rails brought to prominence) and the original implementation in Smalltalk. It has constantly been honed and tweaked since its inception, tailored for the various environments in which it has been used.
Bringing MVC to the Web

Smalltalk’s MVC implementation was created with traditional desktop GUI systems in mind. The separation of responsibilities that it represents makes a lot of sense for web-based software; the model is the representation of the business and persistence layers, the controller is the server-side glue, and the view is the HTML rendered for the client browser.

However, in traditional MVC, the view observes changes in the model in order to reflect its current state by responding to events that the model issues. In a standard HTTP request/response situation, this isn't viable.

Model 2 is a derivative of MVC that was implemented in the Java Struts framework, which introduced a potential solution to this issue. Rather than the view and model directly communicating, the controller becomes a marshaling point for changes. It responds to changes in the view and passes them to the model and vice versa, as shown in the following diagram:

This is the way in which Ruby on Rails implements MVC and in turn inspired a multitude of similar MVC frameworks for the Web (such as ASP.NET MVC).

This is in contrast to web technologies (such as Classic ASP, PHP, and Cold Fusion), where it’s standard practice to create a page that combines logic, rendering, and database access. This can be described (although rarely is) as Model 1 with the MVC implementation as its logical successor. A Model 1 approach leads to the problems, which we described at the beginning of Chapter 1, Introduction, and so the popularization of MVC, in particular the simplified approach that Ruby on Rails took, begins to provide a strong basis for a well-constructed application.

MVC on the Web might follow this request flow:

1. The browser makes a request that is passed on to a controller.
2. The controller consumes the request parameters.
3. It retrieves a model (typically from a database) based on these parameters.
4. Finally, it renders a view based on the model and passes it back to the browser.
MVC and MVVM

Of course, with the advent of Ajax, WebSockets, and fully client-side MVC frameworks, this is a very simplified example. It does serve to show how MVC can easily be adapted for the Web and, in fact, suits the Web very well.

Ext JS and MVC

We've looked at the origins of MVC and the way it was adapted for traditional server-side web applications. How does it work when we use it with the kind of JavaScript-heavy application we'd typically build using Ext JS?

The whole MVC concept moves entirely into the browser. As far as we're concerned, the server can use any technology it wants. It'll generally just provide and consume data to and from the browser. We move back to an MVC implementation that is a little more like the Smalltalk version (different UI elements you see on-screen are views) and each can have their own controller.

Again, this is about breaking down responsibility. Instead of having a single controller take care of an entire page, we can have a search controller, a list controller, and a detail controller (anything that represents the logical units that make up our application). This is a key detail in how the step from server-side MVC to client-side MVC can help our application architecture.

We already know that Ext JS Components are our views, and Ext JS models are well named to fit right in. We're left with one important question: what are controllers actually supposed to do? It's probably easier to remove the things we know they don't do and see what's left. We know that models deal with data, but they're also responsible for the calculations and logic around this data. Calculations and rules, for example, belong in a model, but not in a controller.

This is a generalization. In many cases, you'll have other classes that do this logic work in order to further break down your application. The important thing to take away is that you do not want domain logic in your controller!

We also know views deal with presentation. You could build up an HTML string in your controller and pass it to the browser for rendering, but this would involve the controller in something which is the view's responsibility.

What are we left with? In truth, not much. All your controllers need to do is be in charge of your views and models. That's it. They look at the request the user is making, fetch a model, and use it to render the view to the browser.
In fact, if your controller is doing more than this, you need to take this as a bad sign. A controller should be the conductor of your orchestra, not the one making the music.

**Examples of Ext JS MVC**

The following screenshot shows our Ext JS v4 MVC test application:

![Ext JS MVC Screenshot]

We've generated a stock Ext JS v4 application here, which sticks to the MVC structure and then we've amended it to suit our needs. In this small app, there's a grid of music albums on the left. When you click on the button on the grid, it generates a summary of the artists who are mentioned in the grid, and when you double-click on a row, it puts the album information in the right-hand pane. It's a toy application, but it's useful to demonstrate how MVC works. Later, we'll compare it with a similar application written with Ext JS v5. Let's take a look at the code:

```javascript
// view/List.js
Ext.define('MvcEx1v4.view.List', {
    extend: 'Ext.grid.GridPanel',
    alias: 'widget.app-list',
    store: 'Albums',
    forceFit: true,
    frame: true,
    requires: ['Ext.Msg'],
    columns: [
        { text: 'Name', dataIndex: 'name' },
        { text: 'Artist', dataIndex: 'artist' }
    ],
    initComponent: function() {
        this.bbar = [
```
Here’s our `MvcEx1v4.view.List` class in `view/List.js`. It’s a fairly straightforward grid that uses a store called ‘Albums’ and a button on the bottom toolbar to generate the artist summary. Notice that the event handler to generate this summary is included in the view:

```javascript
Ext.define('MvcEx1v4.view.Detail', {
    extend: 'Ext.Container',
    alias: 'widget.app-detail',
    html: 'Double-click an Album to select'
});
```

Our second view is `MvcEx1v4.view.Detail` in `view/Detail.js`. This is just a simple container with some placeholder HTML. Finally, we have the application viewport that holds our views:

```javascript
Ext.define('MvcEx1v4.view.Viewport', {
    extend: 'Ext.container.Viewport',
    requires: ['MvcEx1v4.view.List'],
    layout: 'hbox',
    defaults: {
        width: 250,
        margin: 20
    },
    items: [{ xtype: 'app-list' }, { xtype: 'app-detail' }]
});
```
Again, there are a few surprises here. Notice that we refer to our views using the values we defined in their individual "alias" configuration options: app-detail and app-list. We've taken care of the "V" in MVC, so let's move on to "M" and see where our data comes from, as shown in the following code:

```javascript
// model/Album.js
Ext.define('MvcEx1v4.model.Album', {
    extend: 'Ext.data.Model',
    fields: [
        { name: 'name', type: 'string' },
        { name: 'artist', type: 'string' }
    ]
});

// store/Albums.js
Ext.define('MvcEx1v4.store.Albums', {
    extend: 'Ext.data.JsonStore',
    model: 'MvcEx1v4.model.Album',
    data: [
        { name: 'In Rainbows', artist: 'Radiohead' },
        { name: 'Swim', artist: 'Caribou' }
    ]
});
```

For easier reading, I've combined the code for the model and the store that consumes it. The data for this application is added inline using the data configuration option (just to avoid messing around with server-side Ajax calls). Let's look at the final facet of MVC, the controller:

```javascript
// controller/Album.js
Ext.define('MvcEx1v4.controller.Album', {
    extend: 'Ext.app.Controller',
    refs: [{
        ref: 'detail',
        selector: 'app-detail'
    }],

    init: function() {
        this.control({
            '.app-list': {
                itemdblclick: this.onAlbumDblClick
            }
        },
    }
});
```
Here's where things start to deviate from the straightforward view to data implementation you'd typically see in an Ext JS v3 application. We're introducing a new class that brings in a new architectural construct. But to what end?

The answer is communication. The controller, as we know, is the glue that sticks together the "M" and the "V". In our simple example here, it's giving us a mechanism to let the list view talk to the detail view without either of them having to be aware of each other. The control feature is used to determine what to do when the list view (aliased as app-list) fires an itemdblclick event.

We supply the onAlbumDblClick method to respond to this event. In here, we want to talk to our detail view (aliased as app-detail). We previously used the refs configuration option to help with this. Let's break it down:

```javascript
refs: [{
  // We give our ref the name "detail". This autogenerates
  // a method on the controller called "getDetail" which
  // will enable us to access the view defined by the selector.
  ref: 'detail',

  // The selector is passed to Ext.ComponentQuery.query,
  // so any valid component query would work here. We're
  // just directly referencing the app-detail alias we
  // set up in the view's configuration
  selector: 'app-detail'
}]
```

Long story short, the refs feature gives us a shorthand way to access a view. In the onAlbumDblClick handler, we use the autogenerated this.getDetail() method that refs provides. This gives us a reference to the view. We can then set HTML of its view's element based on the event data provided by the list view.
How does it help your application

Let's recap. How are we better off than in Ext JS 3 before we had any of this MVC stuff involved?

• We've got clear separation of presentation and data with views and models
• We have a way of orchestrating different parts of our application using controllers
• We've got a way of splitting our app into logic units by using multiple controllers with associated views

Not only does this lend itself to a good design by keeping different bits of functionality very separate from the outset, but it also gives us a good platform for maintainability purposes because it imposes a very specific way of working.

MVC and the illusion of choice

Given everything we've just covered, you'd think that MVC was the holy grail of development. It's tried and tested, adaptable, and supported by Ext JS. In fact, there are some cases in which it's useful to go a little further and augment MVC.

To use Ext JS-specific examples, let's look at what happens when you start writing a more complicated application. Your controllers can react to the events that your views fire, orchestrate interactions between different views, and even stores other controllers. So, does this mean that you put your event handlers in your controllers, your views, or a combination of both?

This is a key question, which can be answered simply by being very strict with your development process from the beginning. MVC provides the "illusion of choice"; in this, it offers a large variety of ways to set up your application, but only a few that will result in a healthy application.

How about when you have a central source of data, but different views that consume it? You might want to have this data in slightly different forms for each view. Does the view itself take responsibility for shaping this data?

Ext JS 5 implements a pattern called model-view-viewmodel (MVVM) that tries to address these questions.
Introducing MVVM

MVVM can be seen as an augmentation of MVC. Introducing the view model concept, it recognizes that not every view concerned with a dataset will be using this data in the same way. It adds a layer of indirection between a view and a model, called a view model, to solve this issue. It also keeps separation of concerns to the fore; why should the model, which is dealing with our data be concerned about anything to do with our view, which is dealing with presentation?

A typical representation of MVVM

How does Ext JS use MVVM?

With Ext JS 5, MVVM is wholeheartedly embraced. The sample application structure that Sencha Cmd generates will provide a `ViewModel` class alongside the `View` class. This has tight integration into the `View` class via new configuration options, which make it a first-class citizen when trying to solve the common problems that arise in a large MVC application, as we discussed earlier.

In addition, a `ViewController` class is created to encapsulate the logic you'd normally put in a view or in a standard controller. It removes the question of where to put event handlers that are concerned with things internal to the view, rather than event handlers that will be passing off events to other parts of your application.

Getting our MVVM started

We started by generating an Ext JS 5 application template using Sencha Cmd and used it as a basis to build an implementation of our example album list application. The default Ext JS 5 template uses an MVVM implementation as follows:
Our example app ported to Ext JS 5's MVVM architecture

The most immediate difference you'll notice is that we've lost our controllers directory and there's a lot more going on in the views directory. Let's break it down:

```javascript
// model/Album.js
Ext.define('MvvmEx1v5.model.Album', {
    extend: 'Ext.data.Model',

    fields: [
        { name: 'name', type: 'string' },
        { name: 'artist', type: 'string' }
    ]
});
```

The album model is identical to the previous example, although note that we've changed the application name to `MvvmEx1v5`. The store is only very slightly different:

```javascript
// store/Albums.js
Ext.define('MvvmEx1v5.store.Albums', {
    extend: 'Ext.data.JsonStore',

    model: 'MvvmEx1v5.model.Album',

    data: [
        { name: 'In Rainbows', artist: 'Radiohead' },
        { name: 'Swim', artist: 'Caribou' }
    ]
});
```
We've added the alias configuration option so that we can refer to the store later using the albums shorthand. Now, let's take a look at the views directory:

```javascript
// view/album/Album.js
Ext.define('MvvmEx1v5.view.album.Album', {
    extend: 'Ext.container.Container',
    xtype: 'app-album',
    requires: ['Ext.grid.Panel'],
    controller: 'album',
    layout: 'hbox',
    defaults: {
        width: 250,
        margin: 20
    },
    items: [
        {
            xtype: 'grid',
            reference: 'list',
            viewModel: 'album',
            bind: '{albums}',
            forceFit: true,
            frame: true,
            margin: '20 10 20 20',
            columns: [
                { text: 'Album', dataIndex: 'name' },
                { text: 'Artist', dataIndex: 'artist' }
            ],
            hbar: [
                '->',
                { xtype: 'button', text: 'Show Artist Summary',
                  handler: 'onShowSummary' },
                '->'
            ],
            listeners: {
                rowdblclick: 'onAlbumDblClick'
            },
        },
    ]
});
```
We've combined the previous app-list and app-detail views into a single app-albums view, and whereas before we had the logic to build the album summary in the view, we now define only the event handler and the logic goes elsewhere. This view is now 100 percent presentation and defers all the complicated stuff to other classes.

Note that we have a controller configuration option that defines the view controller to use for this view class. Our grid component has several interesting configuration options too:

- **reference**: We use this later to get hold of this component from the view controller.
- **viewModel**: This is the alias of the view model this component will use.
- **bind**: This defines how the view talks to the view model. We're using the simplest binding (the grid's default bindProperty is store), so here we're essentially just setting the store config to 'albums'.

Now, let's move on to our album view model:

```javascript
// view/album/AlbumModel.js
Ext.define('MvvmEx1v5.view.album.AlbumModel', {
    extend: 'Ext.app.ViewModel',
    alias: 'viewmodel.album',

    requires: [
        'MvcEx1.store.Albums',
        'Ext.Msg'
    ],

    stores: {
        albums: {
            type: 'albums'
        }
    },

    buildSummary: function() {
        return this.getStore('albums').collect('name').join(', ');
    }
});
```
Also, here we have one of these places that now contain this logic. A view model takes data from a model (or stores) and presents it in a way suitable for its matching view. In this case, we take the data from the 'albums' store (referenced in the type configuration option by the albums alias we mentioned earlier). It provides a buildSummary method that transforms the stored data into a string ready to be used in the UI as follows:

```javascript
// view/album/AlbumController.js
Ext.define('MvvmEx1v5.view.album.AlbumController', {
    extend: 'Ext.app.ViewController',
    alias: 'controller.album',

    onShowSummary: function() {
        var summary =
            this.lookupReference('list').getViewModel().buildSummary();

        Ext.Msg.alert('Artists', summary);
    },

    onAlbumDblClick: function(list, record) {
        var html = Ext.String.format('{0} by {1}',
            record.get('name'), record.get('artist'));
        this.lookupReference('detail').getEl().setHtml(html);
    }
});
```

Finally, we have our view controller, which is where any logic that manages our view should go. Event handlers defined in the view controller will automatically be available to the matching view.

**Are we better off?**

Yes, we're better off because we're more organized. We know where all of our application bits go, and although we've got a lot more files than a straightforward one-class Ext JS application, we'll always know where to look to change the configuration of a view, where to find our logic for the albums, or where to shape the information we pull from a store.

One important point about this example is that we forego the overarching controller from the first example in favor of a view controller. Here, this makes sense; we want this view controller to be concerned only with albums, not with other parts of the application. However, a higher-level controller is still a valid piece of the Ext JS MVVM architecture and can be reintroduced in situations that require a way to coordinate an application at a higher level than a view controller.
A brief interlude regarding stores

Throughout this entire chapter, we've talked a lot about models, but never specifically about stores despite using them in our example applications. Why isn't it "SVC" or "SVVM"?

In Ext JS, a store is a specific class, which brings specific functionality and is tightly bound into your application. However, in a simple MVC or MVVM implementation, the "store" could just be an array of models, rather than being a separate architectural feature. So, a store is really just a way of collecting models together and Ext JS happens to be the place where we can do lots of extra things (such as sorting, filtering, and batching).

Inter-communication

We've shown how to create a simple application that uses several moving parts to create a logical unit. Thanks to the MVVM pattern, we have a methodology that allows the parts of this unit to communicate without having to be explicitly tied to the implementation details of other parts.

When we extend our application, we'll have several of these logical units, perhaps, an artist section in addition to an album section. Now, these have to communicate with each other in turn. This represents one of the main problems in software architecture: how to allow albums and artists to talk to each other without contaminating either component with details of the other. It's a problem that scales in direct proportion to the size and complexity of an application.

The main event

One approach to this problem is to allow your application parts to fire off custom events, each containing a payload that can be consumed by any other part of your application that might be interested in them.

In fact, we see this in action all the time in web development. Event handlers are an integral part of JavaScript programming as we bind functions to the events thrown by user interface elements (such as buttons) or to browser events (such as window.onload). We've touched on this in our example code already; our view fired off a rowdblclick event that was handled by our view controller.

In complex applications, developers will often implement a feature called an event bus, a way of taking the events that application components fire off and transporting them to various subscribers. Since Ext JS 4.2, event domains have allowed developers to incorporate a similar feature into their code base.
Event domains

Event domains allow controllers to react to events from various different sources in your application. The default sources are:

- **Components**: These are events fired from components. This is essentially the functionality that `Ext.app.Controller.control()` provides by handling events from classes that extend `Ext.Component` and bind them to event listeners.
- **Global**: These are events fired from a single global source and used to bind arbitrary application-wide events.
- **Controller**: These are events fired from other controllers.
- **Store**: These are events fired from a store.
- **Direct**: These are events fired from classes that extend `Ext.direct.Provider`. This is only used if you require `Ext.direct.Manager` in your application.

Some event domains allow you to filter the received events by a selector (usually the ID associated with the source), but in the Component case, you can use a full-blown `Ext.Component` query. This allows you to have finer-grained control on how to subscribe to events.

An event domain example

Let's go back to our MVVM album example we created earlier. Our view had handler and listener configurations that tied view events to event handlers that we put in our view controller. However, event domains allow us to remove this tie and give all of the control to the view component. In `view/album/Album.js` in our previous example, we can remove the listener config on the grid and the handler on the button and then add the following code to `view/album/AlbumController.js`:

```javascript
init: function() {
    this.listen({
        component: {
            'app-album grid': {
                'rowdblclick': 'onAlbumDblClick'
            },
            'app-album button': {
                'click': 'onShowSummary'
            }
        }
    });
},
```
This is slightly more verbose, so look at what exactly is happening here. We pass an object to `this.listen`, which contains a component property; this indicates we are configuring the Component Event Domain. Inside here, we use two selectors, one for the grid itself and one for the summary button, and inside these definitions we specify the event we are binding to and the event handlers.

This gives us the ability to remove anything clever from the view and put it all in the view controller. The view deals purely with presentation and the view controller deals with the logic.

**Using custom events**

We've shown how event domains can be used to separate our code concerns even further, but now, you'll see how they can help orchestrate interactions at a higher level. For this, let's take a look at a theoretical situation in which our application has grown to incorporate multiple views and view controllers:

```javascript
// view/search/SearchController.js

  init: function() {
    this.listen({
      component: {
        'app-search button': {
          'click': 'onSearchSubmit'
        }
      }
    });
  },

  onSearchSubmit: function() {
    var val = this.lookupReference('searchfield').getValue();
    this.fireEvent('search', val);
  }
});
```

We've created a new `SearchController`, which is the view controller for a new `Search` view. We use `this.listen` to listen for events on the component event domain and filter them using the selector 'app-search button' (a button within our new `Search` view). When the button is clicked on, we trigger an event handler method called `onSearchSubmit`.

---

[37]
We extract the search term that the user entered and then fire a second event, passing the search term as the event data. The event we fire is called 'search' and rather than being tied to a button or other UI component, it can be subscribed to by other parts of the application as a pure application event. Let's take a look at how it could be consumed:

```javascript
// partial /view/album/AlbumController.js
init: function() {
  this.listen({
    controller: {
      '*': {
        'search': 'onSearch'
      }
    }
  });
}
```

This is a snippet of the AlbumController we've seen before with some extra goodness. With `this.listen`, we use the '*:*' selector to allow all controllers on the event domain. Then, we specify that we want to handle the search event with the `onSearch` handler method. This should all be feeling pretty familiar by now! The handler method could be as simple as the following code:

```javascript
onSearch: function(searchTerm) {
  var list = this.lookupReference('list');
  list.getViewModel().search('searchTerm');
}
```

Assume that we created a `search` method on the view model. With just a small amount of code, we allowed two distinct parts of our application to communicate using information about our application rather than information about themselves. This is key to keeping the search part of this code unaware about the albums part and allows very clear-cut divisions between them. This provides easier testing through separation of concerns, better maintainability, and easier comprehension of how the application is structured.
Summary

MVC and MVVM are key architectural constructs that we need to firmly understand before starting on a new project. Given that they're so embedded in Ext JS, it's even more important to have a good grasp of the ideas behind them and why implementing such patterns will assist in the way we construct our code base. In the next chapter, we'll move on to more practical examples of structuring an Ext JS application, incorporating MVVM concepts with a variety of other ideas that set out a strong platform to build on.
Where to buy this book

You can buy Ext JS Application Development Blueprints from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.