CHAPTER 8

View Models & Data Binding

In this chapter, I am going to introduce you to *view models* and *data binding*. These are two essential techniques that allow you create apps which scale well, are easy to develop and maintain, and respond fluidly to data changes.

You may already be familiar with models and view models from design patterns such as Model-View-Controller (MVC), Model-View-ViewModel (MVVM) and Model-View-View Controller (MVVC). I am not going to get into the details of these patterns in this book. There is a lot of good information about MVC, MVVM and MVVC available, starting with Wikipedia, which has some very balanced and insightful descriptions.

I find the benefits of using a view model to be enormous and well worth considering for all but the simplest app projects, and I recommend you seriously consider following the same path. I am not a pattern zealot, and I firmly believe in taking the parts of patterns and techniques that solve real problems and adapting them to work in specific projects. To that end, you will find that I take a pretty liberal view of how a view model should be used.

The WinJS features that I describe in this chapter underpin some of the fundamental interaction models that Windows app support. To make sure I set a solid foundation for the more advanced features, I start this chapter slowly and gradually introduce the key concepts. Understanding these features is a precursor to getting the best out of advanced UI controls and concepts such as semantic zoom, which I describe in Chapter 16. Table 8-1 provides the summary for this chapter.

*Table 8-1. Chapter Summary*

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
<th>Listing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define and consume a view model</td>
<td>Create a globally available object using the WinJS.Namespace.define method and refer to the object properties as the authoritative source of data.</td>
<td>1-10</td>
</tr>
<tr>
<td>Create observable objects</td>
<td>Use the WinJS.Binding.as method.</td>
<td>11</td>
</tr>
<tr>
<td>Receive change notifications from observable objects</td>
<td>Use the bind method (which is added to the object by the as method)</td>
<td>12</td>
</tr>
<tr>
<td>Manually orchestrate updates for calculated properties in an observable object</td>
<td>Use the notify method to signal a change</td>
<td>13-15</td>
</tr>
<tr>
<td>Activate declarative bindings</td>
<td>Call the WinJS.Binding.processAll method</td>
<td>16</td>
</tr>
</tbody>
</table>
Revisiting the Example App

In this chapter, I continue to build on the AppBars project that I created in the previous chapter. As a reminder, this app introduced NavBars and AppBars and contained a number of simple content pages. I’ll build on this foundation to show new app features.

Decoupling App Components

I am going to start by applying a view model to fix some of the shortcomings of the example app from Chapter 7. In doing this, I will show you my preferred structure for view model objects and demonstrate how simple a view model can be while still making the life of the developer easier.

Note  As I said before, I take a very liberal position on what constitutes a view model and that includes data which is not presented to the user directly.

Defining the View Model

The most important characteristics of a view model are global availability and consistency. In a Windows app, the easiest way to create a basic view model is to use the WinJS.Namespace feature (which I introduced in Chapters 3 and 4) to create and export a view model object to the global namespace. Listing 8-1 shows the contents of the viewmodel.js file, which I added to the js folder of the AppBars example project.

Listing 8-1. The contents of the viewmodel.js file

(function () {
    "use strict";

    WinJS.Namespace.define("ViewModel.State", {
        appBarElement: null,
        navBarContainerElement: null,
        navBarControlElement: null
    });
}());