Summary

ASP.NET MVC is a very good platform for creating unit-testable applications that are maintainable in the long run, as well as for achieving a clearer separation between the UI and the UI-handling logic. In the previous chapter, we learnt the n-tier architecture, but the GUI in such an architecture was still not easily testable as we had a lot of embedded UI code in code-behind classes that was dependent on ViewState and postbacks. But in MVC, the concept of ViewState and postback does not exist as each request is unique in and of itself. This also presents some technical challenges if you have a complex UI, such as a GridView with inline edit or update functionality with a lot of AJAX functionality, where MVC may not work as expected (though the upcoming ASP.NET MVC framework releases will offer more flexible solutions to handle such cases).

Also, server controls such as `DropDownList` may not work as expected in the ASP.NET MVC framework, since there would be no control-level events in the code-behind and the control won't be able to postback. The core principle of the ASP.NET MVC framework is that the URL should talk directly to the requested resource in the web application. This is also the core principle of REST. So if we are using control-level events such as the `selected_change()` method of a `DropDownlist` in code-behind to process some logic, we will be breaking this very REST/MVC principle. The reason is that during such an event, the form will postback without any change in the URL, hence the concept of REST will not hold true (remember that a resource on the web can be accessed by a unique RESTful URL). ASP.NET MVC also has an out-of-the-box powerful URL rewriting capability so that we can have SEO friendly URLs in our application.

This implies that the entire page lifecycle will have no value in the MVC framework. Using MVC means going back to using standard HTML controls instead of rich server controls (even though Microsoft is coming up with a separate set of MVC server controls which will help us in adopting MVC in our applications easily).

The ASP.NET MVC framework was still evolving during the time of writing of this book. There can be a lot of upcoming improvements in the framework in terms of added functionalities and utilities. These changes will cause the final ASP.NET MVC framework release to be somewhat different in terms of syntax shown in this chapter. But the core principle of MVC will remain the same and it will be quite easy to use the latest framework release once we understand the basic principles of the MVC design.

Using MVC in real world commercial applications needs more skill than using the standard ASP.NET postback model. But it gives us the unique benefit of unit testing the GUI layer along with the BL and the DAL, which really helps in Test Driven Development (TDD). So, the ASP.NET MVC framework is not a silver bullet, and you should carefully consider when and in which projects to use it. There will also be a learning curve associated with it for developers coming from a webforms background. It is not a replacement for webforms, but now developers have a choice of using either the MVC or the standard webforms page-centric postback model. ASP.NET MVC can easily be used with an n-tier architecture, as we saw in our code sample.

So ASP.NET MVC is a very good choice for creating a unit-testable and search engine friendly web application which makes our web UI much cleaner by having a clear separation between the UI and code logic.