So the code-behind is kept very light and has no extra code besides data binding to the repeater. Note that usually in the standard postback model, we bind the data under the `if(!IsPostBack)` condition so that data binding happens only once, on the page load, and not on the postback.

Here we cannot follow the same pattern as there is no concept of a postback in MVC. Each request will be as unique as the RESTful URL.

If we want, we can also bind the data in the ASPX using inline code without using code-behind, as shown here:

```
<h2>Customer list</h2>
    <%foreach (var c in ViewData.Model.customers) { %>
        <div>
        <%=c.Name %>
        <br />
        </div>
    <%} %>
```

Note the use of the "var" type to get the `CustomerData` object. This helps us in avoiding explicit casting to get the `Customer` object.

In the same way, we can edit and add objects. An important point to note is that we don't need to use the standard ASP.NET button controls any more, because we don't want the page to postback to itself. Instead, when we add a customer, we can use something like this:

```
    <form method="post" action="Customer/Add">
        <input type="text" name="customerName" value="" />
        <input type="submit" name="Add" value="Add" />
    </form>
```

Notice that there is no `runat="server"` tag here in these controls as they are not server controls but simple HTML controls. On clicking the **Add** button, the page will post with the action `Customer/Add`. This means that, in the `CustomerController`, it will fire the `Add` method as shown in the sample code (just for demonstration purposes):

```
public class CustomerController : Controller
    {
    public ActionResult Add(string customerName)
        {
        //create a business object and fire the add method
        Customer customer = new Customer();
        customer.Name = customerName;
        CustomerCollection customers = new CustomerCollection();
        customers.Add(customer);
        return View("Home");
        }
    }
```

ASP.NET MVC will automatically set the parameter, `customerName`, with the value of the textbox from the form's post data, and will pass this value in the `Add` method of the controller. So we did not have to create the entire page object again, as the page did not postback to the same form. Hence, we avoided recreating the page class on every request or postback.

Because this chapter is more about understanding the MVC framework from the architecture's perspective, we will not go into the details of the edit and add actions; it is best to refer to the following post for these details:

`http://weblogs.asp.net/scottgu/archive/2007/12/09/asp-net-mvc-framework-part-4-handling-form-edit-and-post-scenarios.aspx`

# Unit Testing and ASP.NET MVC

Unit testing is the process where the developer himself tests the code that he has written. The word "unit" here refers to modules of code that he writes, such as methods, functions, and so on. The developer would test such "units" code to verify whether everything is working as expected. This will make sure that at least the individual methods of a class work without errors. Unit testing is different from the function or integration testing that a QA (quality assurance) person performs on a working model after the development phase is over. Unit testing is more closely related to the actual code testing and ensures that most of the bugs are taken care of before the project reaches the actual testing stage. Because unit testing checks the actual methods in the code, it can easily be automated by creating mock test cases in the code using one of the many available unit testing frameworks, such as NUnit and MBUnit.

Earlier in this chapter, we stressed the need for unit testing the GUI of our ASP.NET projects, and how difficult it is to do so under the standard page controller model. But now with ASP.NET MVC, we have "thin" code-behind classes, with almost little or no code. There are almost no Session and ViewState related issues, as all URLs are handled directly by the central controller. There are no button clicks, no code-behind event handlers, and so on. So it is easy to set up unit test cases for the controller classes because then it is the same as testing any normal C# class methods, such as the DAL, the BL, and so on.