

Pro ASP.NET MVC Framework

Copyright © 2009 by Steven Sanderson

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-4302-1007-8

ISBN-13 (electronic): 978-1-4302-1008-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editor: Ewan Buckingham

Technical Reviewer: Andy Olsen

Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell,

Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, Matthew Moodie, Duncan Parkes,

Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh

Project Manager: Sofia Marchant

Copy Editor: Damon Larson

Associate Production Director: Kari Brooks-Copony

Production Editor: Laura Esterman

Compositor: Molly Sharp

Proofreader: Lisa Hamilton

Indexer: BIM Indexing and Proofreading Services

Artist: April Milne

Cover Designer: Kurt Krames

Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit <http://www.springeronline.com>.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit <http://www.apress.com>.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales—eBook Licensing web page at <http://www.apress.com/info/bulksales>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com>.



Your First ASP.NET MVC Application

The best way to appreciate a software development framework is to jump right in and use it. In this chapter, you'll create a simple data entry application using the ASP.NET MVC Framework.

Note In this chapter, the pace is deliberately slow. For example, you'll be given step-by-step instructions on how to complete even small tasks such as adding new files to your project. Subsequent chapters will assume greater familiarity with C# and Visual Studio.

Preparing Your Workstation

Before you can write any ASP.NET MVC code, you need to install the relevant development tools to your workstation. ASP.NET MVC development requires

- Windows XP, Vista, Server 2003, Server 2008, or Windows 7.
- Visual Studio 2008 with SP1 (any edition), or the free Visual Web Developer 2008 Express with SP1. You *cannot* build ASP.NET MVC applications with Visual Studio 2005.

If you already have Visual Studio 2008 with SP1 or Visual Web Developer 2008 Express with SP1 installed, then you can download a stand-alone installer for ASP.NET MVC from www.asp.net/mvc/.

If you don't have either Visual Studio 2008 or Visual Web Developer 2008 Express, then the easiest way to get started is to download and use Microsoft's Web Platform Installer, which is available free of charge from www.microsoft.com/web/. This tool automates the process of downloading and installing the latest versions of any combination of Visual Web Developer Express, ASP.NET MVC, SQL Server 2008 Express, IIS, and various other useful development tools. It's very easy to use—just make sure that you select the installation of both ASP.NET MVC and Visual Web Developer 2008 Express.¹

1. If you use Web Platform Installer 1.0, beware that you must install Visual Web Developer 2008 Express first, and *then* use it to install ASP.NET MVC. It will fail if you try to install both in the same session. This problem is fixed in Web Platform Installer 2.0.

Note While it is possible to develop ASP.NET MVC applications in the free Visual Web Developer 2008 Express (and in fact I've just told you how to install it), I recognize that the considerable majority of professional developers will instead use Visual Studio, because it's a much more sophisticated commercial product. Almost everywhere in this book I'll assume you're using Visual Studio and will rarely refer to Visual Web Developer 2008 Express.

OBTAINING AND BUILDING THE FRAMEWORK SOURCE CODE

There is no technical requirement to have a copy of the framework's source code, but many ASP.NET MVC developers like to have it on hand for reference. While you're in the mood for downloading things, you might like to get the MVC Framework source code from www.codeplex.com/aspnet.

Once you've extracted the source code ZIP file to some folder on your workstation, you can open the solution file, `MvcDev.sln`, and browse it in Visual Studio. You should be able to build it with no compiler errors, and if you have the Professional edition of Visual Studio 2008, you can use **Test ► Run ► All Tests** in Solution to run over 1,500 unit tests against ASP.NET MVC itself.

Creating a New ASP.NET MVC Project

Once you've installed the ASP.NET MVC Framework, you'll find that Visual Studio 2008 offers ASP.NET MVC Web Application as a new project type. To create a new ASP.NET MVC project, open Visual Studio and go to **File ► New ► Project**. Make sure the framework selector (top-right) reads **.NET Framework 3.5**, and select **ASP.NET MVC Web Application**, as shown in Figure 2-1.

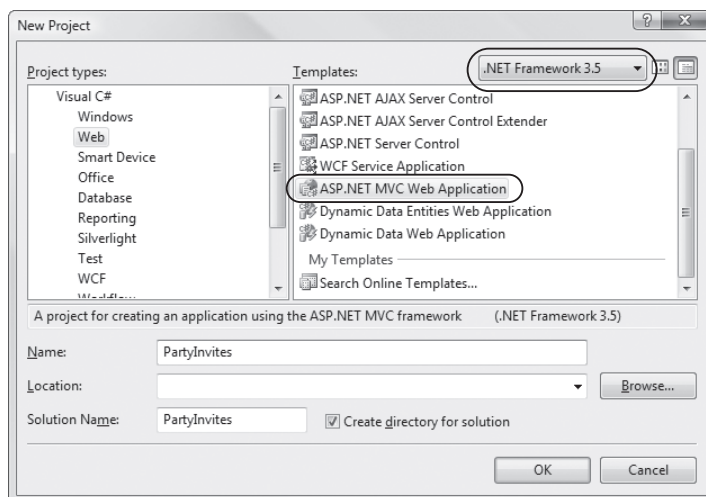


Figure 2-1. Creating a new ASP.NET MVC web application

You can call your project anything you like, but since this demonstration application will handle RSVPs for a party (you'll hear more about that later), a good name would be *PartyInvites*.

When you click OK, the first thing you'll see is a pop-up window asking if you'd like to create a unit test project (see Figure 2-2).

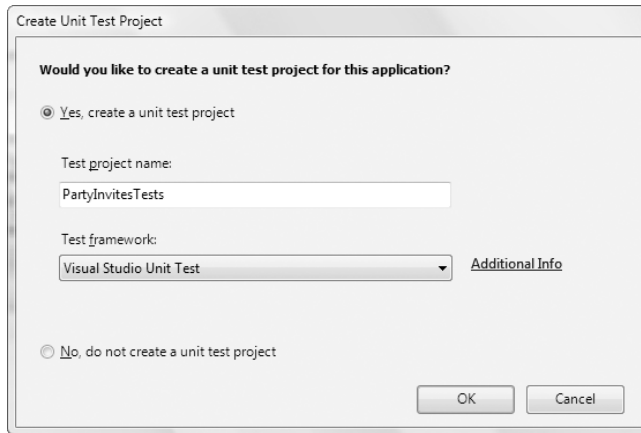


Figure 2-2. Visual Studio prompts to create a unit test project.

For simplicity, we won't write any unit tests for this application (you'll learn more about unit tests in Chapter 3, and use them in Chapter 4). You can choose "No, do not create a unit test project" (or you can choose Yes—it won't make any difference). Click OK.

Visual Studio will now set up a default project structure for you. Helpfully, it adds a default controller and view, so that you can just press F5 (or select Debug ► Start Debugging) and immediately see something working. Try this now if you like (if it prompts you to enable debugging, just click OK). You should get the screen shown in Figure 2-3.

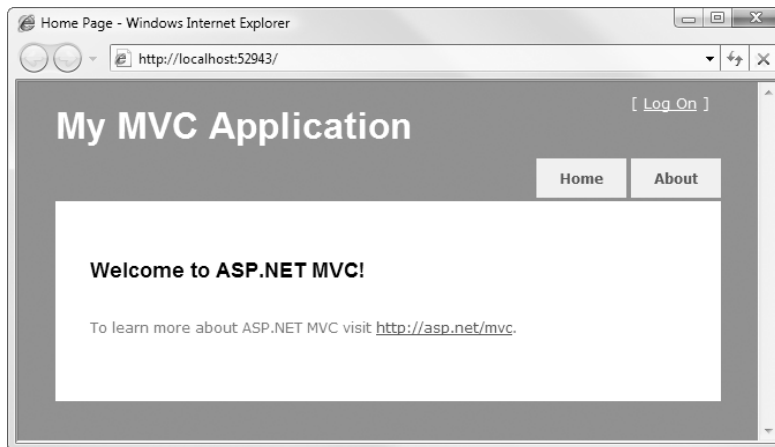


Figure 2-3. The default newborn ASP.NET MVC web application

When you're done, be sure to stop debugging by closing the Internet Explorer window that appeared, or by going back to Visual Studio and pressing Shift+F5 to end debugging.

Removing Unnecessary Files

Unfortunately, in its quest to be helpful, Visual Studio goes a bit too far. It's already created a miniapplication skeleton for you, complete with user registration and authentication. That's a distraction from *really* understanding what's going on, so we're going to delete all that and get back to a blank canvas. Using Solution Explorer, delete each of the files and folders indicated in Figure 2-4 (right-click them, and then choose Delete):

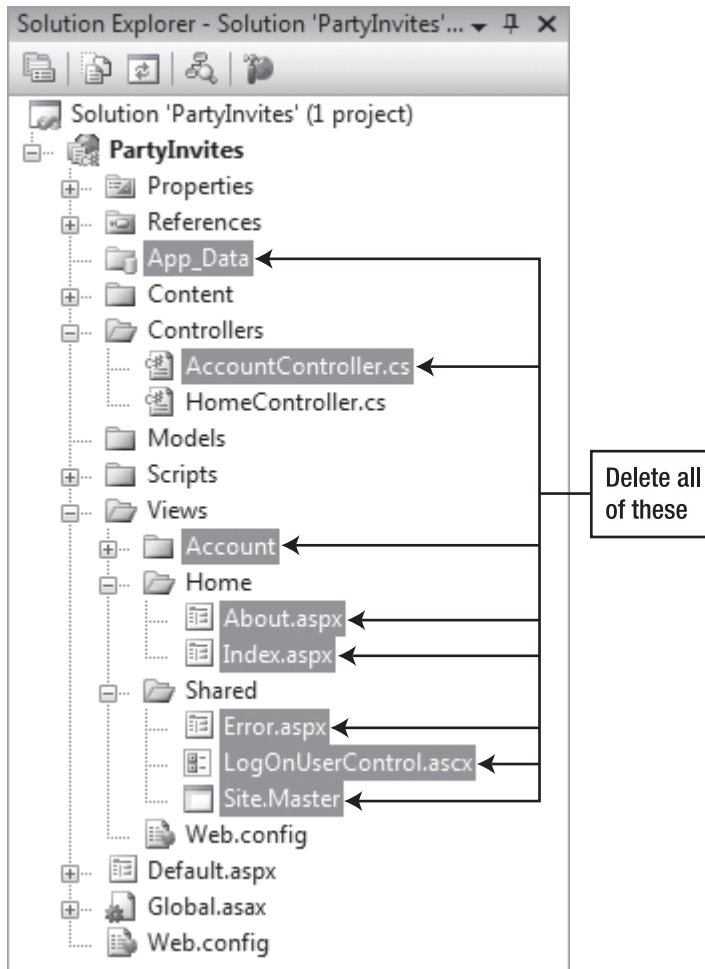


Figure 2-4. Pruning the default project template back to a sensible starting point

The last bit of tidying is inside `HomeController.cs`. Remove any code that's already there, and replace the whole `HomeController` class with this:

```
public class HomeController : Controller
{
    public string Index()
    {
        return "Hello, world!";
    }
}
```

It isn't very exciting—it's just a way of getting right down to basics. Try running the project now (press F5 again), and you should see your message displayed in a browser (Figure 2-5).

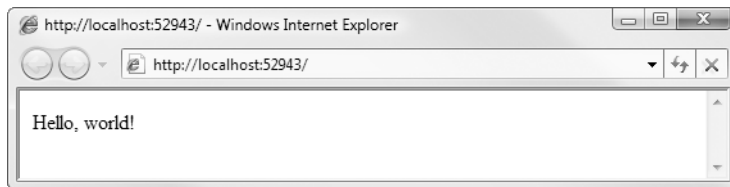


Figure 2-5. *The initial application output*

How Does It Work?

In model-view-controller (MVC) architecture, *controllers* are responsible for handling incoming requests. In ASP.NET MVC, controllers are just simple C# classes² (usually derived from `System.Web.Mvc.Controller`, the framework's built-in controller base class). Each public method on a controller is known as an *action method*, which means you can invoke it from the Web via some URL. Right now, you have a controller class called `HomeController` and an action method called `Index`.

There's also a *routing system*, which decides how URLs map onto particular controllers and actions. Under the default routing configuration, you could request any of the following URLs and it would be handled by the `Index` action on `HomeController`:

- /
- /Home
- /Home/Index

So, when a browser requests `http://yoursite/` or `http://yoursite/Home`, it gets back the output from `HomeController`'s `Index` method. Right now, the output is the string `Hello, world!`.

2. Actually, you can build ASP.NET MVC applications using any .NET language (e.g., Visual Basic, IronPython, or IronRuby). But since C# is the focus of this book, from now on I'll just say "C#" in place of "all .NET languages."

Rendering Web Pages

If you've come this far, well done—your installation is working perfectly, and you've already created a working, minimal controller. The next step is to produce some HTML output.

Creating and Rendering a View

Your existing controller, `HomeController`, currently sends a plain-text string to the browser. That's fine for debugging, but in real applications you're more likely to generate an HTML document, and you do so by using a *view template* (also known simply as a *view*).

To render a view from your `Index()` method, first rewrite the method as follows:

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }
}
```

By returning an object of type `ActionResult`, you're giving the MVC Framework an instruction to render a view. Because you're generating that `ActionResult` object by calling `View()` with no parameters, you're telling the framework to render the action's *default view*. However, if you try to run your application now, you'll get the error message displayed in Figure 2-6.

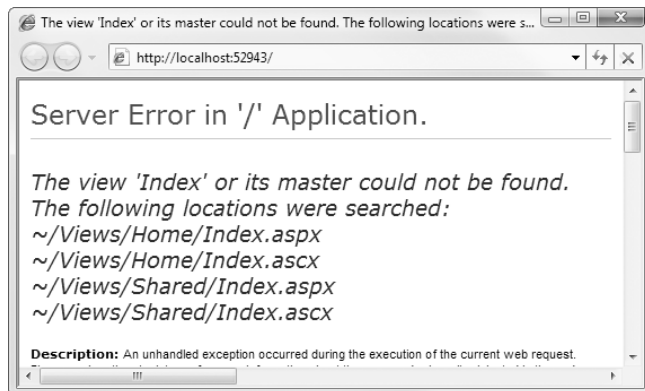


Figure 2-6. Error message shown when ASP.NET MVC can't find a view template

It's more helpful than your average error message—the framework tells you not just that it couldn't find any suitable view to render, but also where it tried looking for one. Here's your first bit of “convention over configuration”: view templates are normally associated with action methods by means of a naming convention, rather than by means of explicit configuration. When the framework wants to find the default view for an action called `Index` on a controller called `HomeController`, it will check the four locations listed in Figure 2-6.

To add a view for the `Index` action—and to make that error go away—right-click the action method (either on the `Index()` method name or somewhere inside the method body) and then choose **Add View**. This will lead to the pop-up window shown in Figure 2-7.

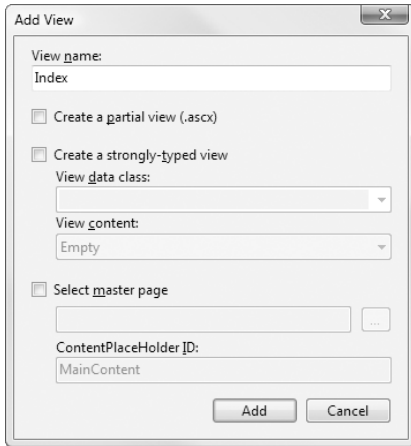


Figure 2-7. Adding a view template for the Index action

Uncheck “Select master page” (since we’re not using master pages in this example) and then click Add. This will create a brand new view template for you at the correct default location for your action method: `~/Views/Home/Index.aspx`.

As Visual Studio’s HTML markup editor appears,³ you’ll see something familiar: an HTML page template prepopulated with the usual collection of elements—`<html>`, `<body>`, and so on. Let’s move the Hello, world! greeting into the view. Replace the `<body>` section of the HTML template with

```
<body>
    Hello, world (from the view)!
</body>
```

Press F5 to launch the application again, and you should see your view template at work (Figure 2-8).

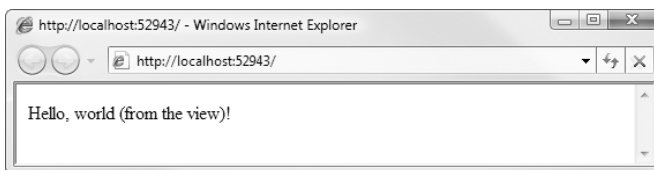


Figure 2-8. Output from the view

Previously, your `Index()` action method simply returned a string, so the MVC Framework had nothing to do but send that string as the HTTP response. Now, though, you’re returning an object of type `ViewResult`, which instructs the MVC Framework to render a view. You didn’t specify a view name, so it picks the conventional one for this action method (i.e., `~/Views/Home/Index.aspx`).

3. If instead you get Visual Studio’s WYSIWYG designer, switch to Source view by clicking Source near the bottom of the screen, or by pressing Shift+F7.

Besides `ViewResult`, there are other types of objects you can return from an action, which instruct the framework to do different things. For example, `RedirectResult` performs a redirection, and `HttpUnauthorizedResult` forces the visitor to log in. These things are called *action results*, and they all derive from the `ActionResult` base class. You'll learn about each of them in due course. This action results system lets you encapsulate and reuse common response types, and it simplifies unit testing tremendously.

Adding Dynamic Output

Of course, the whole point of a web application platform is the ability to construct and display *dynamic* output. In ASP.NET MVC, it's the controller's job to construct some data, and the view's job to render it as HTML. This separation of concerns keeps your application tidy. The data is passed from controller to view using a data structure called `ViewData`.

As a simple example, alter your `HomeController`'s `Index()` action method (again) to add a string into `ViewData`:

```
public ViewResult Index()
{
    int hour = DateTime.Now.Hour;
    ViewData["greeting"] = (hour < 12 ? "Good morning" : "Good afternoon");
    return View();
}
```

and update your `Index.aspx` view template to display it:

```
<body>
    <%= ViewData["greeting"] %>, world (from the view)!
</body>
```

Note Here, we're using *inline code* (the `<%= ... %>` block). This practice is sometimes frowned upon in the ASP.NET WebForms world, but it's your route to happiness with ASP.NET MVC. Put aside any prejudices you might hold right now—later in this book you'll find a full explanation of why, for MVC view templates, inline code works so well.

Not surprisingly, when you run the application again (press F5), your dynamically chosen greeting will appear in the browser (Figure 2-9).

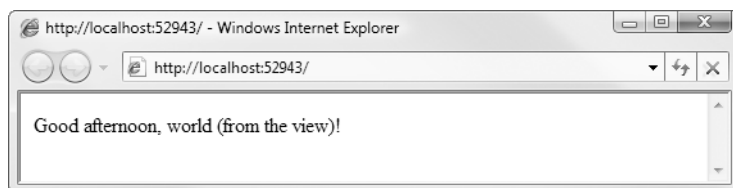


Figure 2-9. Dynamically generated output

A Starter Application

In the remainder of this chapter, you'll learn some more of the basic ASP.NET MVC principles by building a simple data entry application. The goal here is just to see the platform in operation, so we'll create it without slowing down to fully explain how each bit works behind the scenes.

Don't worry if some parts seem unfamiliar to you. In the next chapter, you'll find a discussion of the key MVC architectural principles, and the rest of the book will give increasingly detailed explanations and demonstrations of virtually all ASP.NET MVC features.

The Story

Your friend is having a New Year's party, and she's asked you to create a web site that allows invitees to send back an electronic RSVP. This application, `PartyInvites`, will

- Have a home page showing information about the party
- Have an RSVP form into which invitees can enter their contact details and say whether or not they will attend
- Validate form submissions, displaying a thank you page if successful
- E-mail details of completed RSVPs to the party organizer

I can't promise that it will be enough for you to retire as a Web 3.0 billionaire, but it's a good start. You can implement the first bullet point feature immediately: just add some HTML to your existing `Index.aspx` view:

```
<body>
  <h1>New Year's Party</h1>
  <p>
    <%= ViewData["greeting"] %>! We're going to have an exciting party.
    (To do: sell it better. Add pictures or something.)
  </p>
</body>
```

Linking Between Actions

There's going to be an RSVP form, so you'll need to place a link to it. Update `Index.aspx`:

```
<body>
  <h1>New Year's Party</h1>
  <p>
    <%= ViewData["greeting"] %>! We're going to have an exciting party.
    (To do: sell it better. Add pictures or something.)
  </p>
  <%= Html.ActionLink("RSVP Now", "RSVPForm") %>
</body>
```

Note `Html.ActionLink` is an *HTML helper method*. The framework comes with a built-in collection of useful HTML helpers that give you a convenient shorthand for rendering not just HTML links, but also text input boxes, check boxes, selection boxes, and so on, and even custom controls. When you type `<%= Html.`, you'll see Visual Studio's IntelliSense spring forward to let you pick from the available HTML helper methods. They're each explained in Chapter 10, though most are obvious.

Run the project again, and you'll see the new link, as shown in Figure 2-10.

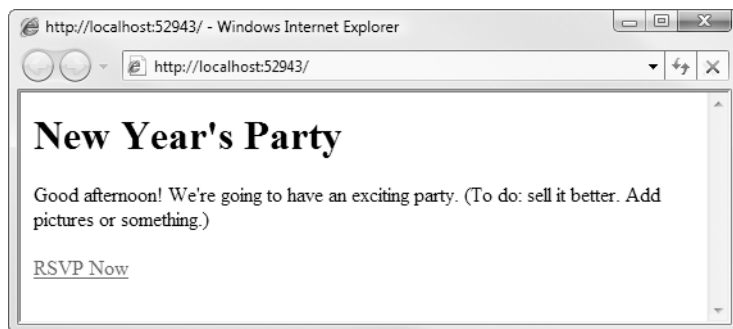


Figure 2-10. *A view with a link*

But if you click the RSVP Now link, you'll get a 404 Not Found error. Check out the browser's address bar: it will read `http://yourserver/Home/RSVPForm`.

That's because `Html.ActionLink` inspected your routing configuration and figured out that, under the current (default) configuration, `/Home/RSVPForm` is the URL for an action called `RSVPForm` on a controller called `HomeController`. Unlike in traditional ASP.NET WebForms, PHP, and many other web development platforms, URLs in ASP.NET MVC *don't* correspond to files on the server's hard disk—instead, they're mapped through a routing configuration on to a controller and action method. Each action method automatically has its own URL; you don't need to create a separate page or class for each URL.

Of course, the reason for the 404 Not Found error is that you haven't yet defined any action method called `RSVPForm()`. Add a new method to your `HomeController` class:

```
public ActionResult RSVPForm()
{
    return View();
}
```

Again, you'll need to add a view for that new action, so right-click inside the method and choose Add View. Uncheck "Select master page" again, and then click Add, and you'll get a new view at this action's default view location, `~/Views/Home/RSVPForm.aspx`. You can leave the

view as is for now, but check when running your application that clicking the RSVP Now link renders your new blank page in the browser.

Tip Practice jumping quickly from an action method to its default view and back again. In Visual Studio, position the caret inside either of your action methods, right-click, and choose Go To View, or press Ctrl+M and then Ctrl+G. You'll jump directly to the action's default view. To jump from a view to its associated action, right-click anywhere in the view markup and choose Go To Controller, or press Ctrl+M and then Ctrl+G again. This saves you from hunting around when you have lots of tabs open.

Designing a Data Model

You could go right ahead and fill in `RSVPForm.aspx` with HTML form controls, but before you do that, take a step back and think about the application you're building.

In MVC, *M* stands for *model*, and it's the most important character in the story. Your *model* is a software representation of the real-world objects, processes, and rules that make up the subject matter, or *domain*, of your application. It's the central keeper of data and domain logic (i.e., business processes and rules). Everything else (controllers and views) is merely plumbing needed to expose the model's operations and data to the Web. A well-crafted MVC application isn't just an ad hoc collection of controllers and views; there's always a model, a recognizable software component in its own right. The next chapter will cover this architecture, with comparisons to others, in more detail.

You don't need much of a domain model for the PartyInvites application, but there is one obvious type of model object that we'll call `GuestResponse`. This object will be responsible for storing, validating, and ultimately confirming an invitee's RSVP.

Adding a Model Class

Use Solution Explorer to add a new, blank C# class called `GuestResponse` inside the `/Models` folder, and then give it some properties:

```
public class GuestResponse
{
    public string Name { get; set; }
    public string Email { get; set; }
    public string Phone { get; set; }
    public bool? WillAttend { get; set; }
}
```

This class uses C# 3 *automatic properties* (i.e., `{ get; set; }`). Don't worry if you haven't caught up with C# 3 yet—the new syntaxes are covered at the end of the next chapter. Also notice that `WillAttend` is a *nullable* `bool` (the question mark makes it nullable). This creates a tri-state value: `True`, `False`, or `null`—the latter value for when the guest hasn't yet specified whether they'll attend.

Building a Form

It's now time to work on `RSVPForm.aspx`, turning it into a form for editing instances of `GuestResponse`. Go back to `RSVPForm.aspx`, and use ASP.NET MVC's built-in helper methods to construct an HTML form:

```
<body>
    <h1>RSVP</h1>

    <% using(Html.BeginForm()) { %>
        <p>Your name: <%= Html.TextBox("Name") %></p>
        <p>Your email: <%= Html.TextBox("Email")%></p>
        <p>Your phone: <%= Html.TextBox("Phone")%></p>
        <p>
            Will you attend?
            <%= Html.DropDownList("WillAttend", new[] {
                new SelectListItem { Text = "Yes, I'll be there",
                                     Value = bool.TrueString },
                new SelectListItem { Text = "No, I can't come",
                                     Value = bool.FalseString }
            }, "Choose an option") %>
        </p>
        <input type="submit" value="Submit RSVP" />
    <% } %>
</body>
```

For each form element, you're specifying a name parameter for the rendered HTML tag (e.g., `Email`). These names match exactly with the names of properties on `GuestResponse`, so by convention ASP.NET MVC associates each form element with the corresponding model property.

I should point out the `<% using(Html.BeginForm(...)) { ... } %>` helper syntax. This creative use of C#'s `using` syntax renders an opening HTML `<form>` tag where it first appears and a closing `</form>` tag at the end of the using block. You can pass parameters to `Html.BeginForm()`, telling it which action method the form should post to when submitted, but since you're not passing any parameters to `Html.BeginForm()`, it assumes you want the form to post to the same URL from which it was rendered. So, this helper will render the following HTML:

```
<form action="/Home/RSVPForm" method="post" >
    ... form contents go here ...
</form>
```

Note “Traditional” ASP.NET WebForms requires you to surround your entire page in exactly one *server-side form* (i.e., `<form runat="server">`), which is WebForms' container for `ViewState` data and postback logic. However, ASP.NET MVC doesn't use server-side forms. It just uses plain, straightforward HTML forms (i.e., `<form>` tags, usually but not necessarily generated via a call to `Html.BeginForm()`). You can have as many of them as you like in a single view page, and their output is perfectly clean—it doesn't add any extra hidden fields (e.g., `__VIEWSTATE`), it doesn't mangle your element IDs, and it doesn't automatically inject any extra JavaScript blocks.

I'm sure you're itching to try your new form out, so relaunch your application and click the RSVP Now link. Figure 2-11 shows your glorious form in all its magnificent, raw beauty.⁴

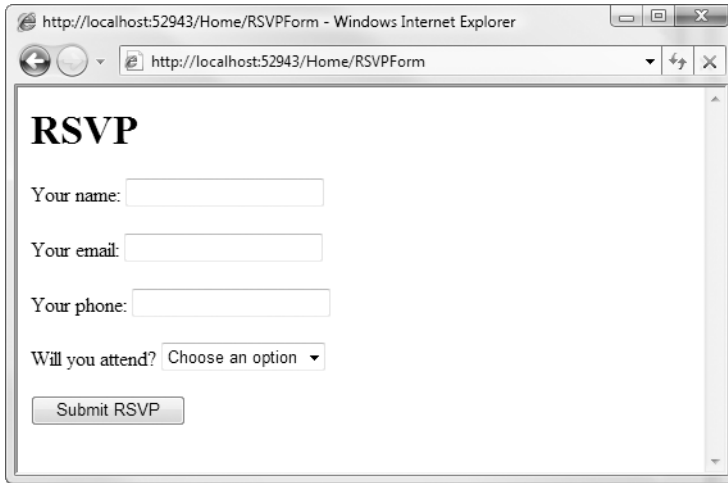
A screenshot of a Windows Internet Explorer browser window. The address bar shows the URL 'http://localhost:52943/Home/RSVPForm'. The page content is titled 'RSVP' in a large, bold, serif font. Below the title are four form elements: a text input field labeled 'Your name:', a text input field labeled 'Your email:', a text input field labeled 'Your phone:', and a dropdown menu labeled 'Will you attend?' with the text 'Choose an option' and a downward arrow. At the bottom of the form is a button labeled 'Submit RSVP'.

Figure 2-11. Output from the *RSVPForm.aspx* view

Dude, Where's My Data?

If you fill out the form and click Submit RSVP, a strange thing will happen. The same form will immediately reappear, but with all the input boxes reset to a blank state. What's going on? Well, since this form posts to `/Home/RSVPForm`, your `RSVPForm()` action method will run again and will render the same view again. The input boxes will be blank because there isn't any data to put in them—any user-entered values will be discarded because you haven't done anything to receive or process them.

Caution Forms in ASP.NET MVC do not behave like forms in ASP.NET WebForms! ASP.NET MVC deliberately does not have a concept of “postbacks,” so when you rerender the same form multiple times in succession, you shouldn't automatically expect a text box to retain its contents. In fact, you shouldn't even think of it as being the same text box on the next request: since HTTP is stateless, the input controls rendered for each request are totally newborn and independent of any that preceded them. However, when you do want the effect of preserving input control values, that's easy, and we'll make that happen in a moment.

4. This book isn't about CSS or web design, so we'll stick with the retro chic “Class of 1996” theme for most examples. ASP.NET MVC values pure, clean HTML, and gives you total control over your element IDs and layouts, so you'll have no problems using any off-the-shelf web design template or fancy JavaScript effects library.

Handling Form Submissions

To receive and process submitted form data, we're going to do a clever thing. We'll slice the `RSVPForm` action down the middle, creating

One method that responds to HTTP GET requests: Note that a GET request is what a browser issues normally each time someone clicks a link. This version of the action will be responsible for displaying the initial blank form when someone first visits `/Home/RSVPForm`.

Another method that responds to HTTP POST requests: By default, forms rendered using `Html.BeginForm()` are submitted by the browser as a POST request. This version of the action will be responsible for receiving submitted data and deciding what to do with it.

Writing these as two separate C# methods helps keep your code tidy, since the two methods have totally different responsibilities. However, from outside, the pair of C# methods will be seen as a single logical action, since they will have the same name and are invoked by requesting the same URL.

Replace your current single `RSVPForm()` method with the following:

```
[AcceptVerbs(HttpVerbs.Get)]
public ActionResult RSVPForm()
{
    return View();
}

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult RSVPForm(GuestResponse guestResponse)
{
    // Todo: Email guestResponse to the party organizer
    return View("Thanks", guestResponse);
}
```

Tip You'll need to import the `PartyInvites.Models` namespace; otherwise, Visual Studio won't recognize the type `GuestResponse`. The least brain-taxing way to do this is to position the caret on the unrecognized word, `GuestResponse`, and then press `Ctrl+dot`. When the prompt appears, press `Enter`. Visual Studio will automatically import the correct namespace for you.

No doubt you can guess what the `[AcceptVerbs]` attribute does. When present, it restricts which type of HTTP request an action will respond to. The first `RSVPForm()` overload will only respond to GET requests; the second `RSVPForm()` overload will only respond to POST requests.

Introducing Model Binding

The first overload simply renders the same default view as before. The second overload is more interesting because it takes an instance of `GuestResponse` as a parameter. Given that the method is being invoked via an HTTP request, and that `GuestResponse` is a .NET type that is totally unknown to HTTP, how can an HTTP request possibly supply a `GuestResponse`

instance? The answer is *model binding*, an extremely useful feature of ASP.NET MVC whereby incoming data is automatically parsed and used to populate action method parameters by matching incoming key/value pairs with the names of properties on the desired .NET type.

This powerful, customizable mechanism eliminates much of the humdrum plumbing associated with handling HTTP requests, letting you work primarily in terms of strongly typed .NET objects rather than low-level fiddling with `Request.Form[]` and `Request.QueryString[]` dictionaries as is often necessary in WebForms. Because the input controls defined in `RSVPForm.aspx` have names corresponding to the names of properties on `GuestResponse`, the framework will supply to your action method a `GuestResponse` instance already fully populated with whatever data the user entered into the form. Handy!

Introducing Strongly Typed Views

The second overload of `RSVPForm()` also demonstrates how to render a specific view template that doesn't necessarily match the name of the action, and how to pass a single, specific model object that you want to render. Here's the line I'm talking about:

```
return View("Thanks", guestResponse);
```

This line tells ASP.NET MVC to find and render a view called `Thanks`, and to supply the `guestResponse` object to that view. Since this all happens in a controller called `HomeController`, ASP.NET MVC will expect to find the `Thanks` view at `~/Views/Home/Thanks.aspx`, but of course no such file yet exists. Let's create it.

Create the view by right-clicking inside an action method in `HomeController` and then choosing `Add View`. This time, the view will be slightly different: we'll specify that it's primarily intended to render a single specific type of model object, rather than the previous views which just rendered an ad hoc collection of things found in the `ViewData` structure. This makes it a *strongly typed view*, and you'll see the benefit of it shortly.

Figure 2-12 shows the options you should use in the `Add View` pop-up. Enter the view name `Thanks`, uncheck "Select master page," and this time, check the box labeled "Create a strongly typed view." In the "View data class" drop-down, select the `GuestResponse` type. Leave "View content" set to `Empty`. Finally, click `Add`.

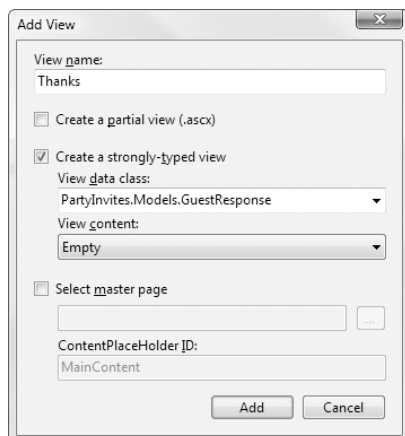


Figure 2-12. Adding a strongly typed view to work with a particular model class

Once again, Visual Studio will create a new view template for you at the location that follows ASP.NET MVC conventions (this time, it will go at `~/Views/Home/Thanks.aspx`). This view is strongly typed to work with a `GuestResponse` instance, so you'll have access to a variable called `Model`, of type `GuestResponse`, which is the instance being rendered. Enter the following markup:

```
<body>
    <h1>Thank you, <%= Html.Encode(Model.Name) %>!/</h1>
    <% if(Model.WillAttend == true) { %>
        It's great that you're coming. The drinks are already in the fridge!
    <% } else { %>
        Sorry to hear you can't make it, but thanks for letting us know.
    <% } %>
</body>
```

The great benefit of using strongly typed views is that not only are you being precise about what type of data the view renders, you'll also get full IntelliSense for that type, as shown in Figure 2-13.

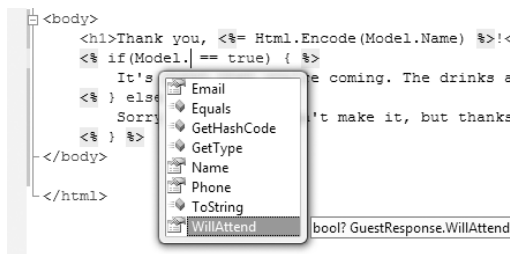


Figure 2-13. Strongly typed views offer IntelliSense for the chosen model class.

You can now fire up your application, fill in the form, submit it, and see a sensible result, as shown in Figure 2-14.



Figure 2-14. Output from the *Thanks.aspx* view

Tip Protect your application from cross-site scripting attacks by HTML-encoding any user input that you echo back. For example, `Thanks.aspx` contains `<%= Html.Encode(Model.Name) %>`, not just `<%= Model.Name %>`. You'll learn more about this and other security matters in Chapter 13.

Adding Validation

You may have noticed that so far, there's no validation whatsoever. You can type in any nonsense for an e-mail address, or even just submit a completely blank form.

It's time to rectify that, but before you go looking for the validation controls, remember that this is an MVC application, and following the don't-repeat-yourself principle, validation is a *model* concern, *not* a UI concern. Validation often reflects business rules, which are most maintainable when expressed coherently in one and only one place, not scattered variously across multiple controller classes and ASPX and ASCX files. Also, by putting validation right into the model, you ensure that its data integrity is always protected in the same way, no matter what controller or view is connected to it. This is a more robust way of thinking than is encouraged by WebForms-style `<asp:XYZValidator>` UI controls.

There are lots of ways of accomplishing validation in ASP.NET MVC. The following technique is perhaps the simplest one, though it's not as tidy or as powerful as some alternatives you'll learn about later in this book. Go and edit your `GuestResponse` class, making it implement the interface `IDataErrorInfo` as follows. I'll omit a full explanation of `IDataErrorInfo` at this point—all you need to know right now is that it simply provides a means of returning a possible validation error message for each property.

```
public class GuestResponse : IDataErrorInfo
{
    public string Name { get; set; }
    public string Email { get; set; }
    public string Phone { get; set; }
    public bool? WillAttend { get; set; }

    public string Error { get { return null; } } // Not required for this example

    public string this[string propName]
    {
        get {
            if ((propName == "Name") && string.IsNullOrEmpty(Name))
                return "Please enter your name";
            if ((propName == "Email") && !Regex.IsMatch(Email, ".+\\@.+\\.\\.+.+"))
                return "Please enter a valid email address";
            if ((propName == "Phone") && string.IsNullOrEmpty(Phone))
                return "Please enter your phone number";
            if ((propName == "WillAttend") && !WillAttend.HasValue)
                return "Please specify whether you'll attend";
            return null;
        }
    }
}
```

Note You'll need to add `using` statements for `System.ComponentModel` and `System.Text.RegularExpressions`. Again, Visual Studio can do this for you with the `Ctrl+dot` trick.

If you're a fan of elegant code, you might want to use a validation framework that lets you collapse all this down to just a few C# attributes attached to properties on the model object (e.g., [ValidateEmail]).⁵ But for this small application, the preceding technique is simple and readable enough.

ASP.NET MVC automatically recognizes the `IDataErrorInfo` interface and uses it to validate incoming data when it performs model binding. Let's update the second `RSVPForm()` action method so that if there are any validation errors, it redisplay the default view instead of rendering the Thanks view:

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult RSVPForm(GuestResponse guestResponse)
{
    if (ModelState.IsValid) {
        // Todo: Email guestResponse to the party organizer
        return View("Thanks", guestResponse);
    }
    else // Validation error, so redisplay data entry form
        return View();
}
```

Finally, choose where to display any validation error messages by adding an `Html.ValidationSummary()` to the `RSVPForm.aspx` view:

```
<body>
    <h1>RSVP</h1>

    <%= Html.ValidationSummary() %>

    <% using(Html.BeginForm()) { %>
        ... leave rest as before ...
```

And now, if you try to submit a blank form or enter invalid data, you'll see the validation kick in (Figure 2-15).

Model Binding Tells Input Controls to Redisplay User-Entered Values

I mentioned previously that because HTTP is stateless, you shouldn't expect input controls to retain state across multiple requests. However, because you're now using model binding to parse the incoming data, you'll find that when you redisplay the form after a validation error, the input controls *will* redisplay any user-entered values. This creates the appearance of controls retaining state, just as a user would expect. It's a convenient, lightweight mechanism built into ASP.NET MVC's model binding and HTML helper systems. You'll learn about this mechanism in full detail in Chapter 11.

5. Such a validation framework could let you avoid hard-coding validation error messages, and allow you to internationalize them, too. You'll learn more about this in Chapter 11.

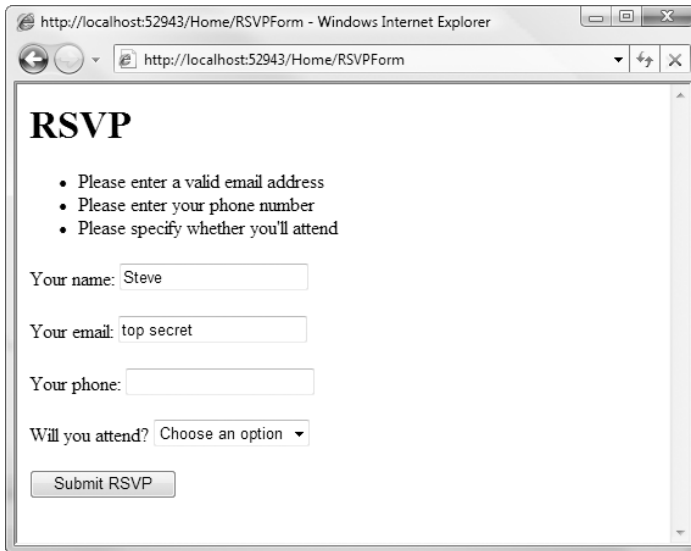
A screenshot of a Windows Internet Explorer browser window. The address bar shows 'http://localhost:52943/Home/RSVPForm'. The page title is 'RSVP'. Below the title, there are three bullet points: 'Please enter a valid email address', 'Please enter your phone number', and 'Please specify whether you'll attend'. Below these are four form fields: 'Your name:' with the value 'Steve', 'Your email:' with the value 'top secret', 'Your phone:' which is empty, and 'Will you attend?' with a dropdown menu showing 'Choose an option'. At the bottom is a 'Submit RSVP' button.

Figure 2-15. *The validation feature working*

Note If you've worked with ASP.NET WebForms, you'll know that WebForms has a concept of "server controls" that retain state by serializing values into a hidden form field called `__VIEWSTATE`. Please rest assured that ASP.NET MVC model binding has absolutely *nothing* to do with WebForms concepts of server controls, postbacks, or ViewState. ASP.NET MVC never injects a hidden `__VIEWSTATE` field—or anything of that sort—into your rendered HTML pages.

Finishing Off

The final requirement is to e-mail completed RSVPs to the party organizer. You could do this directly from an action method, but it's more logical to put this behavior into the model. After all, there could be other UIs that work with this same model and want to submit `GuestResponse` objects. Add the following methods to `GuestResponse`:⁶

```
public void Submit()
{
    EnsureCurrentlyValid();

    // Send via email
    var message = new StringBuilder();
    message.AppendFormat("Date: {0:yyyy-MM-dd hh:mm}\n", DateTime.Now);
```

6. You'll need to add `using System;`, `using System.Net.Mail;`, and `using System.Text;`, too (e.g., by using the Ctrl+dot technique again).

```

message.AppendFormat("RSVP from: {0}\n", Name);
message.AppendFormat("Email: {0}\n", Email);
message.AppendFormat("Phone: {0}\n", Phone);
message.AppendFormat("Can come: {0}\n", WillAttend.Value ? "Yes" : "No");

SmtpClient smtpClient = new SmtpClient();
smtpClient.Send(new MailMessage(
    "rsvps@example.com",                                // From
    "party-organizer@example.com",                       // To
    Name + (WillAttend.Value ? " will attend" : " won't attend"), // Subject
    message.ToString()                                   // Body
));
}

private void EnsureCurrentlyValid()
{
    // I'm valid if IDataErrorInfo.this[] returns null for every property
    var propsToValidate = new[] { "Name", "Email", "Phone", "WillAttend" };
    bool isValid = propsToValidate.All(x => this[x] == null);
    if (!isValid)
        throw new InvalidOperationException("Can't submit invalid GuestResponse");
}

```

If you're unfamiliar with C# 3's lambda methods (e.g., `x => this[x] == null`), then be sure to read the last part of Chapter 3, which explains them.

Finally, call `Submit()` from the second `RSVPForm()` overload, thereby sending the guest response by e-mail if it's valid:

```

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult RSVPForm(GuestResponse guestResponse)
{
    if (ModelState.IsValid)
    {
        guestResponse.Submit();
        return View("Thanks", guestResponse);
    }
    else // Validation error, so redisplay data entry form
        return View();
}

```

As promised, the `GuestResponse` model class protects its own integrity by refusing to be submitted when invalid. A solid model layer shouldn't simply trust that the UI layer (controllers and actions) will always remember and respect its rules.

Of course, it's more common to store model data in a database than to send it by e-mail, and in that case, model objects will normally ensure their validity before they go into the database. The major example in Chapter 4 will demonstrate one possible way to use ASP.NET MVC with SQL Server.

CONFIGURING SMTPCLIENT

This example uses .NET's `SmtpClient` API to send e-mail. By default, it takes mail server settings from your `web.config` file. To configure it to send e-mail through a particular SMTP server, add the following to your `web.config` file:

```
<configuration>
  <system.net>
    <mailSettings>
      <smtp deliveryMethod="Network">
        <network host="smtp.example.com"/>
      </smtp>
    </mailSettings>
  </system.net>
</configuration>
```

During development, you might prefer just to write mails to a local directory, so you can see what's happening without having to set up an actual mail server. To do that, use these settings:

```
<configuration>
  <system.net>
    <mailSettings>
      <smtp deliveryMethod="SpecifiedPickupDirectory">
        <specifiedPickupDirectory pickupDirectoryLocation="c:\email" />
      </smtp>
    </mailSettings>
  </system.net>
</configuration>
```

This will write `.eml` files to the specified folder (here, `c:\email`), which must already exist and be writable. If you double-click `.eml` files in Windows Explorer, they'll open in Outlook Express or Windows Mail.

Summary

You've now seen how to build a simple data entry application using ASP.NET MVC, getting a first glimpse of how MVC architecture works. The example so far hasn't shown the power of the MVC framework (e.g., we skipped over routing, and there's been no sign of automated testing as yet). In the next two chapters, you'll drill deeper into what makes a good, modern MVC web application, and you'll build a full-fledged e-commerce site that shows off much more of the platform.

