With MvcContrib, NHibernate, and more

ASP.NET MVC IN ACTION

Jeffrey Palermo
Ben Scheirman
Jimmy Bogard

FOREWORD BY PHIL HAACK

SAMPLE CHAPTER

MANNING

ASP.NET MVC in Action

by Jeffrey Palermo
Ben Scheirman
and Jimmy Bogard

Chapter 12

brief contents

12 Best practices

This chapter covers

- Designing maintainable controllers, filters, and actions
- Building maintainable views with minimum duplication
- Designing and testing routes
- Testing MVC components

Although the ASP.NET MVC Framework is young in the .NET space, the MVC pattern applied to web applications is not. We have presented thus far techniques already used in many other MVC Frameworks, but some areas in the ASP.NET MVC Framework require extra attention. The ASP.NET MVC Framework is open-ended and extensible for customization, but not all usage and customization is appropriate. Not every approach to solving a problem will lead to elegant, maintainable results. Many of the examples on the web work well for simple problems, but break down quickly in a large production application or slightly complex small applications. In this chapter, we examine the major feature areas and extension points of ASP.NET MVC to discover what parts to use, what parts to avoid, and how to get the most out of our design by following best practices.

12.1 Controllers

As the entry point for the main control of a request, controllers that lack careful design can quickly resemble their predecessors (the old `Page_Load` event) in complexity and opacity. Duplication shows up early in an application with even the slightest complexity, and the best approach is to remove different kinds of duplication. Multiple copy-and-paste commands add up and obscure the true intent of an action method. Using the wrong technique to remove duplication can wind up hiding the mechanism of a request and leading to difficulty in maintenance and troubleshooting.

The designers of ASP.NET MVC studied many mature MVC frameworks to determine the best method of providing extension points and removing duplication. When a request comes in from the outside world, we might have a group of actions we want to restrict to certain roles. Or entire areas of our application might require an authenticated user. Patterns, extensions, and techniques can help a great deal in such situations. In this section, we look at ways to make our controllers easier to develop and maintain. The first of these patterns is the Layer Supertype, applied to controllers.

12.1.1 Layer Supertype

As an application grows, patterns emerge in your controllers—a filter applied to a common group of controllers, or a set of views that need the same data. Because a controller is just a class, nothing technically stops us from using additional supertypes between our concrete controller and the `Controller` type. In these cases, we can employ the Layer Supertype pattern. As the name suggests, a Layer Supertype is a supertype (base class) for all types in that layer. This supertype contains all the common behavior that applies across an entire layer.

In CodeCampServer, the Layer Supertype is the `SmartController` type, shown in figure 12.1.

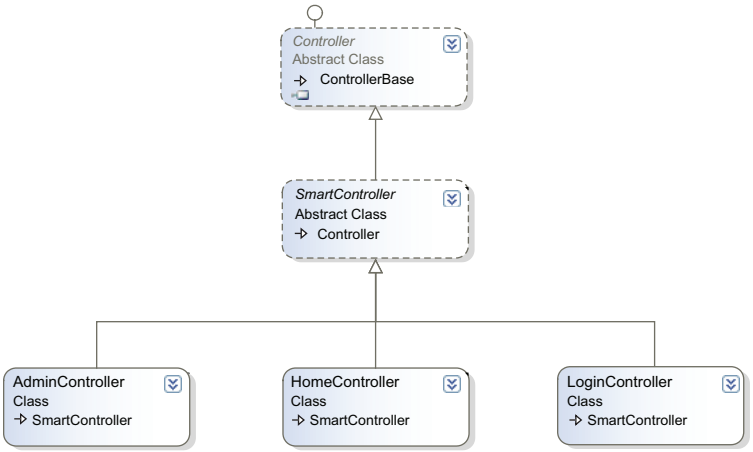


Figure 12.1 The Layer Supertype `SmartController` serving as our new base `Controller` class

One popular use for a Layer Supertype in an MVC application is to gather common filters in one location. Because filters applied to a base type at the class level also apply to derived types, a Layer Supertype is a great choice to use as a common filter. The `SmartController` uses several filters:

- Authentication filter
- Url referrer filter
- Assembly version filter
- User group filter

To display the view, the authentication filter adds the current user in session to `ViewData`. Athough we want this common behavior across many actions and views, it would be painful to put this filter on every controller. We would likely forget some controllers. In addition, maintenance of a filter declared on a large group of controllers is duplication that should be avoided. The other filters perform similar operations, and they place relevant information into `ViewData` so that the views can use the information.

A Layer Supertype is trivial to create: you declare a new controller, mark it as abstract, and place it between the concrete controller type and the MVC base `Controller` type. Any new controller you create can then inherit directly from our Layer Supertype, as shown in listing 12.1.

Listing 12.1 CodeCampServer Layer Supertype and a derived controller

```
public class ScheduleController : SmartController

[AuthenticationFilter]
public abstract class SmartController : Controller
```

Instead of always beginning an application using the Layer Supertype pattern, keep in mind that it is most appropriate for larger applications or complex layers within an application. Trivial applications often do not need this pattern, but CodeCampServer is an example of an application that benefits from it. Every concrete controller has the `SmartController` in its inheritance chain. In addition to common filters, Layer Supertypes also hold common helper methods or properties.

12.1.2 Filters

As described in chapter 3, filters allow us to implement common behavior among many actions or controllers. Not all common behavior belongs in a filter. Filters are only one way of eliminating duplication in ASP.NET MVC. Filters are best at eliminating duplication in action methods and ensuring common behavior is executed among common controllers and actions.

In many web applications, access to certain areas is restricted to authenticated users. If anonymous users attempt to access these areas, they are redirected from these secured resources. To create this authentication, we need two items. First we need an `IAuthorizationFilter` implementation to perform the filter behavior. Second, we need an implementation of the `AuthorizeAttribute` to trigger our custom behavior. The custom filter is shown in listing 12.2.

Listing 12.2 Our custom authorization filter implementation

```
public class EnforceAuthenticationFilter : IAuthorizationFilter
{
    private readonly IUserSession _userSession;     ❶

    public EnforceAuthenticationFilter(IUserSession userSession)
    {
        _userSession = userSession;
    }

    public void OnAuthorization(AuthorizationContext filterContext)
    {
        if (_userSession.GetCurrentUser() == null)
        {
            filterContext.Result = new RedirectResult ("/");     ❷
        }
    }
}
```

Our authentication filter depends on an `IUserSession` ❶, whose implementation is merely a wrapper around `Session`. We need to implement one `IAuthorizationFilter` method—`OnAuthorization`. In this method, the `AuthorizationContext` gives us access to contextual data about the request and inherits from `ControllerContext`. The `AuthorizationContext` also gives us a `Result` property that allows us to provide an optional `ActionResult` to execute. In the previous example, we redirect the user to the home page by setting the `Result` property to a new `RedirectResult` ❷. This happens only when we don't have a current user. Otherwise, the filter does nothing, and the request proceeds as normal.

To apply this filter to our controllers and actions, we need an `AuthorizeAttribute` implementation, shown in listing 12.3.

Listing 12.3 Custom authorization filter delegating to the filter implementation

```
public class EnforceAuthenticationAttribute : AuthorizeAttribute
{
    public override void OnAuthorization(
        AuthorizationContext filterContext)
    {
        var authFilter = IoC.Resolve<EnforceAuthenticationFilter>();

        authFilter.OnAuthorization(filterContext);
    }
}
```

The `EnforceAuthenticationAttribute` inherits from `AuthorizeAttribute`, overriding the `OnAuthorization` behavior to instantiate and call into our `EnforceAuthenticationFilter` for the actual behavior we want. The final step is to apply our new attribute to the controllers we want to enforce authentication on, as shown in listing 12.4.

Listing 12.4 Decorating our controller with the custom authorization attribute

```
[EnforceAuthentication]
public class SecuredController : Controller
{
```

With this filter applied, we prevent anonymous users from taking any action on our `SecuredController`, as well as any derived controllers. How do we know when it's necessary to create a filter? Common, orthogonal behavior fits best into action filters. In listing 12.3, authorization happens for a wide variety of controllers and actions, but this behavior often does not relate to the other behavior in an action. This orthogonal behavior is best abstracted through a filter. In other cases, copying and pasting the same code between actions is another sign that a filter might be appropriate.

Filters can be overused and misapplied, especially as their behavior is not as explicit as code inside an action method. But applied correctly, filters are a great way to add site-wide behavior with little code and little impact on your controllers.

Why separate filters from attributes?

Many examples show the work of the filter being done directly inside of the filter attribute. The custom filter attribute class is the only piece required to implement a custom filter, but we often create a separate filter from the attribute because of the nature and limitations of attributes. If you choose to use a container or factory to locate dependencies, you will not have control over the instantiation of your attribute class. One common workaround is to define two constructors, one that takes the filter's dependencies, and one that calls into the container or factory to supply implementations at runtime. This leads to nontrivial bugs, as your attribute's constructor can be called at nonobvious times, such as in reflection scenarios used by unit testing frameworks.

If your filter does not use dependencies, having two separate classes is overkill. But as soon as you start including dependencies in your filter, take the extra step of separating the attribute from the filter.

12.1.3 Smart binders

The model binders in ASP.NET MVC are useful out of the box. They do a great job of taking request and form input and hydrating fairly complex models from them. But a custom binder can also remove another common form of duplication—loading an object from the database based on an action parameter. Most of the time, this action parameter is the primary key of the object or another unique identifier. Instead of putting this repeated data access code in all of our actions, we can use a custom model binder that can load the persisted object before the action is executed. Our action can then take the persisted object type as a parameter instead of the unique identifier.

One problem with the MVC model binder implementation is that we can match our custom model binder for a single type. In an application with dozens of entities, it

is easy to forget to register the custom model binder for every type. For example, CodeCampServer uses a common base type (`PersistentObject`) for all entities in the system. Ideally, we could register the custom model binder just once, or just leave it up to each custom binder to decide whether or not it should bind.

To accomplish this, we need to replace the default model binder with our own implementation. Additionally, we can define an interface, `IFilteredModelBinder`, for our new binders, as shown in listing 12.5.

Listing 12.5 The `IFilteredModelBinder` interface

```
public interface IFilteredModelBinder : IModelBinder
{
    bool IsMatch(Type modelType);
}
```

The `IFilteredModelBinder` inherits from the MVC `IModelBinder` interface, and adds a method through which implementations can perform custom matching logic. In our case, we can look at the base type of the model type passed in to determine if it is a `PersistentObject` type. To use custom filtered model binders, we need to create a `DefaultModelBinder` implementation, as shown in listing 12.6.

Listing 12.6 A smarter model binder

```
public class SmartBinder : DefaultModelBinder
{
    private readonly IFilteredModelBinder [] _filteredModelBinders;

    public SmartBinder (
       params IFilteredModelBinder [] filteredModelBinders)     ❶
    {
        _filteredModelBinders = filteredModelBinders;
    }

    public override object BindModel (          ❷
        ControllerContext controllerContext,
        ModelBindingContext bindingContext)
    {
        foreach (var modelBinder in _filteredModelBinders)     ❸
        {
            if (modelBinder.IsMatch(bindingContext.ModelType))
            {
                return modelBinder.BindModel (controllerContext,
                    bindingContext);                                ❹
            }
        }

        return base.BindModel (controllerContext, bindingContext);
    }
}
```

Our new `SmartBinder` class takes an array of `IFilteredModelBinders` ❶, which we'll fill in soon. Next, it overrides the `BindModel` method ❷. `BindModel` loops through all of the supplied `IFilteredModelBinders`, and checks to see if any match the `ModelType`

from the `ModelBindingContext` ❸. If it is a match, we execute and return the result from `BindModel` for that `IFilteredModelBinder` ❹. The complete class diagram is shown in figure 12.2.

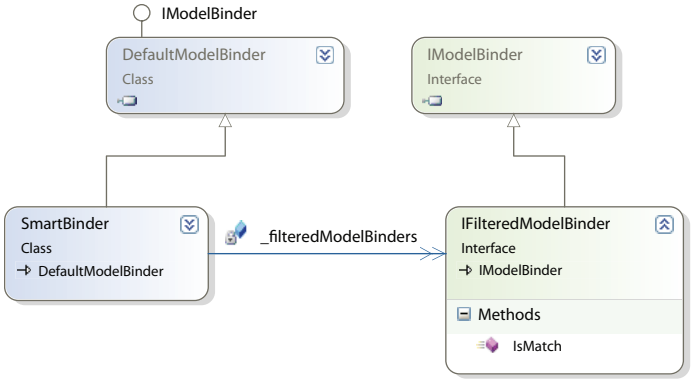


Figure 12.2 The class diagram of our `SmartBinder` showing the relationship to `IFilteredModelBinder`

Now that we have a new binder that can match on more than one type, we can turn our attention to our new model binder for loading persistent objects. This new model binder will be an implementation of the `IFilteredModelBinder` interface. It will need to do a number of things in order to return the correct entity from our persistence layer:

1. Retrieve the request value from the binding context
2. Deal with missing request values
3. Create the correct repository
4. Use the repository to load the entity, and return it

We won't cover the third item in much depth, as this example assumes that an IoC container is in place. The entire model binder needs to implement our `IFilteredModelBinder`, and is shown in listing 12.7.

Listing 12.7 The `EntityModelBinder`

```
public class EntityModelBinder : IFilteredModelBinder
{
    public bool IsMatch(Type modelType)     ❶
    {
        return typeof(PersistentObject).IsAssignableFrom(modelType);
    }

    public object BindModel (
        ControllerContext controllerContext,
        ModelBindingContext bindingContext)
    {
        ValueProviderResult value =
            bindingContext.ValueProvider [bindingContext.ModelName];     ❷

        if (value == null)
            return null;
```

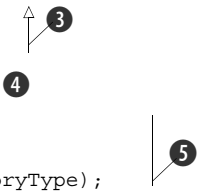
❸

```
        if (string.IsNullOrEmpty(value.AttemptedValue))
            return null;

        var entityId = new Guid(value.AttemptedValue);

        Type repositoryType = typeof(IRepository<>)
            .MakeGenericType(bindingContext.ModelType);
        var repository = (IRepository)IoC.Resolve(repositoryType);
        PersistentObject entity = repository.GetById(entityId);

        return entity;
    }
}
```



In listing 12.7 we implement our newly created interface, `IFilteredModelBinder`. The additional method, `IsMatch` ❶, returns true when the model type being bound by ASP.NET MVC is a `PersistentObject`, our base type for all model objects persisted in a database. Next, we have to implement the `BindModel` method by following the steps laid out just before listing 12.7. First, we retrieve the request value from the `ModelBindingContext` ❷ passed in to the `BindModel` method. The `ModelBindingContext` contains a dictionary of strings to `ValueProviderResults` in the `ValueProvider` property. If the `ValueProviderResult` does not exist, or the attempted value does not exist, we won't try to retrieve the entity from the repository ❸. Although the entity's identifier is a `Guid`, the attempted value is a string, so we construct a new `Guid` from the attempted value on the `ValueProviderResult` ❹.

Now that we have the parsed `Guid` from the request, we can create the appropriate repository from our IoC container ❺. But because we have specific repositories for each kind of entity, we don't know the specific repository type at compile time. However, all of our repositories implement a common interface, as shown in listing 12.8.

Listing 12.8 The common repository interface

```
public interface IRepository<TEntity>
    where TEntity : PersistentObject
{
    TEntity GetById(Guid id);
}
```

We want the IoC container to create the correct repository given the type of entity we are attempting to bind. This means we need to figure out and construct the correct `Type` object for the `IRepository` we create. We do this by using the `Type.MakeGenericType` method to create a closed generic type from the open generic type `IRepository<>`.

Open and closed generic types

An open generic type is simply a generic type that has no type parameters supplied. `IList<>` and `IDictionary<,>` are both open generic types. To create instances of a type, we must create a closed generic type from the open generic type. A closed generic type is a generic type with type parameters supplied, such as `IList<int>` and `IDictionary<string, User>`.

When we have the closed generic type for `IRepository` using the `ModelBindingContext.ModelType` property, we can use our IoC container to create an instance of the repository to call and use. Finally, we call the repository's `GetById` method and return the retrieved entity from `BindModel`. Because we cannot call a generic method at runtime without using reflection, we use another nongeneric `IRepository` interface that returns only objects as `PersistentObject`, as shown in listing 12.9.

Listing 12.9 The nongeneric repository interface

```
public interface IRepository
{
    PersistentObject GetById(Guid id);
}
```

All repositories in our system inherit from a common repository base class, which implements both the generic and nongeneric implementations of `IRepository`. Because some places cannot hold references to the generic interface (as we encountered with model binding) the additional nongeneric `IRepository` interface supports these scenarios.

We have our `SmartBinder` and `EntityModelBinder`, which bind to entities from request values, but we still need to configure ASP.NET MVC to use these binders instead of the default model binder. To do this, we set the `ModelBinders.Binders.DefaultBinder` property in our application startup code, as shown in listing 12.10.

Listing 12.10 Replacing the default model binder

```
protected void Application_Start()
{
    ModelBinders.Binders.DefaultBinder =
        new SmartBinder (new EntityModelBinder ());
```

At this point, we have only a single filtered model binder. In practice, we might have specialized model binders for certain entities, classes of objects (such as enumeration classes), and so on. By creating a model binder for entities, we can create controller actions that take entities as parameters, as opposed to just a `Guid`, as shown in listing 12.11.

Listing 12.11 Controller action with an entity as a parameter

```
public class ConferenceController : Controller
{
    public ViewResult Show(Conference c)
    {
        return View(c);
    }
}
```

With the `EntityModelBinder` in place, we avoid repeating code in our controller actions. This repetition would obscure the intent of the controller action with data access code that is not relevant to what the controller action is trying to accomplish.

12.1.4 Hardcoded strings

ASP.NET MVC often uses dictionaries to pass information and data among different layers. With all these dictionaries floating around, the possibility for hardcoded, duplicated strings increases. Dictionaries are useful when we cannot specify an exact contract for the data to pass between two objects. For example, the controller uses `ViewData` to pass information to the view. But `ViewData` at its heart is nothing more than an enhanced dictionary. If we hardcode our strings in the controller, such as in the listing 12.12 example, we run the risk of brittle and unmaintainable code.

Listing 12.12 Hardcoded string in a controller

```
public ActionResult Register()
{
    ViewData ["PasswordLength"] = MembershipService.MinPasswordLength;

    return View();
}
```

In this case, the `ViewData` key, `"PasswordLength"`, is used no less than six times in the entire sample ASP.NET MVC project! If we ever decided to change this key to something else, we could attempt a global find and replace, but we still run the risk of changing the wrong keys. Repeating this key could cause problems for future maintainers, who might not understand where the key is used. So you wind up with a key value that is never changed, for fear of breaking the application.

Instead, with some static fields and a little organization, we can make sure that this key appears once as a string in our application, as shown in listing 12.13.

Listing 12.13 Organized view data keys

```
public static class Keys
{
    public static class ViewData
    {
        public static readonly string PasswordLength = "PasswordLength";
    }
}
```

When we want to use a certain dictionary key, we go to the one place in our application where these keys are defined. The changed controller action is shown in listing 12.14.

Listing 12.14 Using a single location for a dictionary key

```
public ActionResult Register_after()
{
    ViewData[Keys.ViewData.PasswordLength] =
        MembershipService.MinPasswordLength;

    return View();
}
```

In our view, we'll use the same reference to ensure that we have the key value only once in our application. If we decide to rename the value of this key, or its reference,

we won't break any existing controllers, views, or tests. Hardcoded strings work well, until the values are repeated. Some developers go as far as changing their IDE font color settings for strings to be bright and recognizable, as a visual cue to the developer indicating use of hardcoded strings.

12.1.5 Separated view models

In many applications, the model we pass to the view is our domain model. A screen to show conference information might need only a single `Conference` object to display what it needs. In other cases, we might need more information than what is available on our domain model. A search results screen might need paging information, the number of results returned, and perhaps a list of matched terms. On an edit screen, we may want to include validation on our objects. If users can enter quantities in their shopping carts, how can we ensure they entered valid numbers?

Let's take a simple example: when editing a conference, we are required to enter the maximum number of attendees. This would be represented on our domain model as shown in listing 12.15.

Listing 12.15 Conference model

```
public class Conference : PersistentObject
{
    public int MaxAttendees { get; set; }
}
```

When it comes to binding this to our edit view, we'll have a slight problem. If we make our `Conference` the same model as the one our edit view uses, what happens if a user enters an invalid numerical value, or nothing at all? Model binding either won't work, or will only bind the default to our `MaxAttendees` property. In order to properly handle all use cases, our `MaxAttendees` property can only be a string. This is a case of the view's concerns leaking into our domain model, a situation to be avoided if all possible.

The other option is to create a separate view model type for our views from the domain model. We'll create a specialized class, just for that one view. We can shape that type however we like, and allow the view to shape our view model however we want. The advantage of a separated view model is that our views won't influence the domain model in any way. For less complex applications, this separation is not necessary and overcomplicates the design. As complexity of the views increases, the design of the views has more and more impact on our domain model, unless the view model and domain model are separated.

But the information from our domain still needs to get to the views, and to do this, we can simply transform our domain model into the view model, as shown in listing 12.16.

Listing 12.16 Transforming the domain model to view model

```
public ActionResult Edit(Conference conference)
{
    var editModel = new EditConference
```

```
    {
        Id = conference.Id,
        MaxAttendees = conference.MaxAttendees.ToString()
    };
    return View(editModel);
}
```

Our `Edit` action, which is used to edit a `Conference`, takes a `Conference` domain object as a parameter (using the model binding techniques of the last section). Next, we create an `EditConference` view model object from our `Conference`. In our `EditConference` type, `MaxAttendees` is a string, so we can properly handle malformed input. Finally, we return a `ViewResult` with the `EditConference` object as our model for that view. To handle the final form post for editing, our `Edit` POST action now takes the `EditConference` as a parameter, as shown in listing 12.17.

Listing 12.17 The edit action for posting conference updates

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(EditConference conference)
{
```

One interesting side effect of crafting custom view models is that our views and controllers start to become much more strongly typed. Instead of receiving a bag of form variables, we have a single object to deal with. Instead of myriad dictionary key-value pairs, we have properties on a real type. We do have to write code to process the `EditConference` object and map it to our `Conference` object, but this is something we would need to do in any case. Using an OOM (object-object mapper) such as AutoMapper can save you from writing the lion's share of mapping code. By separating our `EditConference` type from the `Conference` type, we allow each to grow independently, with the only coupling between the two being our mapping code. The view can shape the `EditConference` model, and our domain can shape the `Conference` model. In practice, we found that our domain model and `view models` start with a similar shape, but start to diverge as our views became more and more complex. As we introduced master pages, partials, AJAX, caching, and so on, a separated view model allowed us to contain the complexity of our views, without affecting the design of our domain model.

12.1.6 Validation

Because validation can take so many forms, it's a tricky subject. For a single screen for adding a conference, you might have to enforce several rules:

- Maximum attendees must be a number.
- Maximum attendees is required.
- Maximum attendees cannot exceed what the building's fire code allows.
- A large conference cannot be cancelled without approval of a majority of the organizers.

As we can see, there is no black-and-white distinction among different rules. We must enforce the rule that the "maximum attendees" value entered by the user is a number, but what about it being a required field? That starts to bleed into business rules validation. Validation and business rules include a wide spectrum, from invariants such as type checking and required fields, all the way to complex workflow logic. Validation frameworks work well with the UI end of the spectrum, where we are validating user input for further business rule checking. This distinction is not altogether concrete, especially when we want to tie business rule violations to specific user interface elements.

But validation frameworks help to consolidate and standardize user input validation, providing a common interface for executing and reporting validation errors. There are many validation frameworks in .NET, from the Validation Application Block from the Microsoft Patterns and Practices group, to Data Annotations, originating from ASP.NET Dynamic Data, to open source offerings such as Castle Validators. Each is configured through attributes.

We can use the Castle Validators for data type and required field validation, as shown in listing 12.18.

Listing 12.18 Our `EditConference` model with validation attributes applied

```
public class EditConference
{
    public Guid Id { get; set; }

    [ValidateNonEmpty]
    [ValidateInteger]
    public string MaxAttendees { get; set; }
```

Castle Validators won't execute automatically as part of an ASP.NET MVC request, unless we specifically code them to do so. But we don't want to have to code this validation for every controller action that needs validation. Instead, we can modify our custom model binder to do the validation. When our controller action processes the bound model, it can examine `ModelState` to determine if any validation errors occurred, and take appropriate action. To add our Castle Validator logic to model binding, we'll need to override the `OnModelUpdated` method, as shown in listing 12.19.

Listing 12.19 The modified default model binder with Castle Validator

```
protected override void OnModelUpdated (
    ControllerContext controllerContext,
    ModelBindingContext bindingContext)
{
    var runner = new ValidatorRunner (new CachedValidationRegistry());    (1)
    if (!runner.IsValid(bindingContext.Model))                             (2)
    {
        var summary = runner.GetErrorSummary(bindingContext.Model);        (3)

        foreach (var invalidProperty in summary.InvalidProperties)
        {
```

```
                foreach (var error in summary
                    .GetErrorsForProperty(invalidProperty))
                {
                    bindingContext.ModelState
                        .AddModelError(invalidProperty, error);     ❹
                }
            }
        }
    }
```

To run validation against our model, we'll first need to create the `ValidatorRunner` ❶, the class responsible for executing the validation logic. The `IsValid` method ❷ takes our model, and returns a boolean indicating whether or not validation was successful. If not, we call the `GetErrorSummary` method ❸, looping through the invalid properties and errors for each property, adding the individual error message to the `ModelState` errors collection ❹. Castle Validators have their own structure for representing validation errors, which we have to translate into ASP.NET MVC's mechanism for representing model errors. In our `Edit` action that accepts the form post, we need only to inspect the `ModelState` for any errors, and show the original edit view if validation failed, as shown in listing 12.20.

Listing 12.20 Handling validation errors in our controller action

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(EditConference conference)
{
    if (!ModelState.IsValid)
        return View(conference);
```

In our view, we can use the normal `HtmlHelper` methods to display any validation errors or summaries, as we are using the built-in `ModelState` for Castle Validator validation messages. With our custom model binder, we were able to provide seamless integration between a third-party validation framework and ASP.NET MVC. Our views have no knowledge of any validation framework underneath, allowing us to mix and match validation frameworks to our needs. In the next section, we'll examine ways to ensure our views are easy to deal with in complex and long-term maintenance scenarios.

12.2 Views

It is easy to underestimate the effort needed to create and maintain the HTML markup and code in an MVC application. Although the view is responsible *only* for creating and displaying HTML, duplication exists just as much in views as it would in other layers in our application. We'll look at techniques for reducing duplication as well as using lambda expressions and strongly typed views to eliminate magic strings.

12.2.1 Strongly typed views

Here is one view, one model. When a view accepts only one model type, it allows us to make optimizations along that vector, such as the generic `HtmlHelper` and expression-based HTML generation. Strongly typed views are straightforward to use: when we use

the `View` method in an action method, we pass in the model object. On the view side, we can inherit from the generic `System.Web.Mvc.ViewPage<TModel` base type. Moving away from the dictionary-based view model to a strongly typed model with real properties eliminates many of the pesky runtime errors that crop up due to relying on a weak contract of dictionary keys, which are not compile-safe. With dictionary keys, we have to rely on the strength (and correctness) of our controller unit tests to ensure our views don't break. Strongly typed views also bring another developer convenience: IntelliSense. When we access the Model property in a view, IntelliSense can pick up all members from the underlying model type.

Strongly typed views don't force us away from the dictionary-based `ViewData`. Both can exist in a single view, though these occurrences should be the exception, not the rule. We'll have to handle some scenarios slightly differently, such as filters used for populating `ViewData`. With a strongly typed view, we want to limit the filters being used in this manner. Instead, we can use the `RenderAction` method inside of a view to grab common information that is not already in the existing view model. It may seem strange to call another action inside of a view, but this provides a way for the template of a page to organize other information that may need to appear on many pages. When we use filters for populating `ViewData`, we are introducing a strong, but not obvious, coupling from the view to the controller. When we look in a view and see access to `ViewData` through a key-value pair, it is difficult to trace back and understand where this value came from, and why. With strongly typed views and `RenderAction`, we can enforce strongly typed views throughout our application, regardless of how we organize our views into partials.

Current ASP.NET MVC demos and examples rely heavily on dictionaries to pass data around. Overreliance on these mechanisms can lead to a brittle application, and just the introduction of compile-time safety in strongly typed views is a welcome safety net for reducing bugs.

12.2.2 Fighting duplication

When using Web Forms, we had many different avenues for reducing duplication and providing common visual components. Many of these same avenues exist for MVC; some are new, and some are deprecated. There are at least four ways to consolidate HTML in ASP.NET MVC:

1. Master pages
2. Partials
3. RenderAction
4. Extending `HtmlHelper`

In addition to these four, there are more exotic ways to render HTML in a view, such as subcontrollers and partial requests. The most common mechanisms are the four listed, and each has its sweet spot.

For site-wide layout, master pages are the ideal choice. Using content placeholders, we can create a common layout for our site, and individual pages can insert content for

each placeholder. Master pages can be nested, allowing us to construct general layouts, as well as layouts for sections of a website. The administrative section might have the same header and footer, but a different toolbar and main section layout, whereas the rest of the site might have a preference-generated toolbar, with a sidebar for bookmarks. With master pages, interaction with `ViewData` should be kept to a minimum, as interactions would create a coupling from controller design to view design. Master page design should not affect controller/action design.

Every time you start to copy and paste HTML, stop yourself. This could be an opportunity to refactor that common markup into a partial. Partials, which can themselves be strongly typed, are perfect for small, repeated bits of markup used in more than one view. Partials have their own `ViewData`, which must be supplied by a parent view, and therefore the calling action. If this information is already available in the single view model given to the view through `ViewData`, partials are an appropriate mechanism for consolidating HTML. If you must resort to filters to supply a partial's `ViewData`, your controllers will start to become more and more intimate with the design of your view.

`RenderAction` is an excellent alternative to using the combination of a filter and a partial, because it effectively displays markup and data that is completely orthogonal to your main controller action. A logon widget displayed on every page needs login information. But if your main controller action is displaying a list of products, it muddies the concern of our controller to include the name of the current user. We can put the name of the current user in `ViewData` through a filter, but our view is no longer completely strongly typed. On the view side, we'll see a call to `RenderPartial` and passing in the user's name. Using this method, you will likely not be able to understand from where the information is coming. If you have to search for usages of a dictionary key value, that is likely an indication of too much indirection. With `RenderAction`, the view creates a minipipeline, calling an action with all of the action's filters executed. The choice of locating data for the subview is relegated to the other controller.

We have the choice of extending `HtmlHelper`. The choice between using a partial and extending `HtmlHelper` is fairly clear. If the piece of HTML is only one or two elements, the `HtmlHelper` is an ideal choice. A common scenario for input elements is to include a label alongside the form element. This repetitive HTML gets tedious to write over and over. We can encapsulate this into a single `HtmlHelper` extension, as shown in listing 12.21.

Listing 12.21 Extending `HtmlHelper` to include labels

```
public static string TextBoxWithLabelFor<TModel, TProperty>(
    this HtmlHelper<TModel> htmlHelper,
    Expression<Func<TModel, TProperty>> expression,
    string label)
    where TModel : class
{
    string labelHtml =
```

```
        string.Format("<label for=\"{0}\">{1}:</label>",          ❶
            ExpressionHelper.GetInputName(expression),          ❷
            label);
    string textboxHtml = htmlHelper.TextBoxFor(expression);

    return labelHtml + " " + textboxHtml;
}
```

In our extension, we craft custom HTML for the label ❶, but lean on helpers from the MVC Futures assembly to assist in creating both the textbox and the value of the label's `for` attribute ❷. The markup then becomes much easier to read, as shown in listing 12.22.

Listing 12.22 Using the `HtmlHelper` extension in a view

```
<% using (Html.BeginForm()) { %>
    <div>
        <%= Html.TextBoxWithLabelFor (c => c.MaxAttendees, "Max Attendees")
        %>
        <%= Html.ValidationMessageFor(c => c.MaxAttendees) %>
        <p>
            <input type="submit" value="Submit" />
        </p>
    </div>
<% } %>
```

The MVC Futures assembly contains quite a few extensions that are useful with strongly typed views. We can combine and extend the provided extensions in new and interesting ways. We could add support for validation error messages, asterisks for required fields, or custom HTML for certain kinds of output. We could create extensions for dates to include a calendar picker, or autocomplete functionality for user pickers. Code looks bad in markup, and markup is hard to distinguish in code. If you find yourself piecing together lots of HTML in an `HtmlHelper` extension, you may want to look at partials. If your partial is small, or contains logic, an `HtmlHelper` can provide a cleaner mechanism for the containing view.

To fully take advantage of views in ASP.NET MVC, it helps to understand the different ways we can refactor, extend, and improve how we craft HTML. Not all mechanisms are appropriate in every scenario, and often, the choice between options is not always clear. Views should never get so complicated that it is difficult to move from a partial to an `HtmlHelper` extension or a `RenderAction` call to a filter and a partial. In the next section, we'll examine ways to reduce the number of magic strings present in our application through use of expressions.

12.2.3 Embracing expressions

Magic strings permeate the ASP.NET MVC Framework. They are magic because they represent a property or a well-known attribute, but there is no connection to the original information. When using magic strings, you are prone to the following types of errors: misspelling, refactoring, and renaming bugs. Even the default template included in ASP.NET MVC is riddled with magic strings. Because C# is a statically typed

language, we can leverage the type and symbol checking if we use symbols instead of strings to make relationships among code. As an exercise, let's examine one view in the sample, the LogOn.aspx page, in listing 12.23.

Listing 12.23 Magic strings in a view

```
<h2>Log On</h2>
<p>
    Please enter your username and password.
    <%= Html.ActionLink ("Register", "Register") %>     ❶
    if you don't have an account.
</p>
<%= Html.ValidationSummary("Login was unsuccessful. Please correct the errors
     and try again.") %>

<% using (Html.BeginForm()) { %>
    <div>
        <fieldset>
            <legend>Account Information</legend>
            <p>
                <label for="username">Username:</label>     ❷
                <%= Html.TextBox("username") %>         ❸
                <%= Html.ValidationMessage("username") %>     ❹
            </p>
            <p>
                <label for="password">Password:</label>
                <%= Html.Password("password") %>
                <%= Html.ValidationMessage("password") %>
            </p>
            <p>
                <%= Html.CheckBox("rememberMe") %>
                <label class="inline" for="rememberMe">Remember me?</label>
            </p>
            <p>
                <input type="submit" value="Log On" />
            </p>
        </fieldset>
    </div>
<% } %>
```

We can see several examples of magic strings:

- The `ActionLink` call refers to a method on a controller ❶, and will break if the method name changes.
- All labels refer to a parameter name on a method ❷, which will break if the name changes.
- All input elements refer to a parameter name on a method ❸.
- All validation messages refer to a parameter name on a method ❹.
- Labels, input elements and validation messages must line up, or the page will break.

Fighting these magic string errors is easy–don't use them! Instead, we can use expression-based methods and strongly typed views. Consider the action method that would

accept a form post from the page using the view in listing 12.23. One approach is for the action method to accept each form field as an individual parameter. Another approach would be to create a single view model type to represent the entire form, as shown in listing 12.24.

Listing 12.24 Our new view model form type

```
public class LogOnForm
{
    public string Username { get; set; }
    public string Password { get; set; }
    public bool RememberMe { get; set; }
    public string ReturnUrl { get; set; }
}
```

Our view can then be changed to a strongly typed view for our `LogOnForm`. All HTML generation can now use expressions pointing to a property instead of magic strings, as shown in listing 12.25.

Listing 12.25 Using expressions in our view

```
<h2>Log On</h2>
<p>
    Please enter your username and password.
    <%= Html.ActionLink<AccountController>(c => c.LogOn_after(null),
        "Register") %>                                                    (1)
    if you don't have an account.
</p>
<%= Html.ValidationSummary("Login was unsuccessful. Please correct the errors
     and try again.") %>

<% using (Html.BeginForm()) { %>
    <div>
        <fieldset>
            <legend>Account Information</legend>
            <p>
                <%= Html.TextBoxWithLabelFor(
                    form => form.Username, "Username") %>     (2)
            </p>
            <p>
                <%= Html.PasswordWithLabelFor(
                    form => form.Password, "Password") %>     (3)
            </p>
            <p>
                <%= Html.CheckBoxWithLabelFor(
                    form => form.RememberMe, "Remember me") %>     (4)
            </p>
            <p>
                <input type="submit" value="Log On" />
            </p>
        </fieldset>
    </div>
<% } %>
```

We have introduced many changes in this view:

- The `ActionLink` call now refers to a method on a controller ❶.
- The username input element is generated from an expression ❷.
- The password input element is generated from an expression ❸.
- The "remember me" input element is generated from an expression ❹.

Despite all these changes, our final HTML has not changed. However, our view is now much less susceptible to subtle bugs, which can only be caught at runtime. With strongly typed views and expression-based HTML generation methods, we can feel comfortable that refactoring will work properly and view compilation will fail if we rename or remove a controller action or `view model` property. Not all instances of magic string usage are eliminated in the MVC Futures assembly. Additional extensions are easily added as necessary if they don't already exist in the MvcContrib project. For instance the MvcContrib.FluentHtml assembly can assist with HTML generation.

Magic strings are insidious sources of runtime bugs, and should be eliminated wherever you find them. Common usages of magic strings come in the form of representing method, property, or parameter names as strings in your application. These should be red flags for developers, and replaced with compile-safe, IntelliSense, and refactoring-friendly expressions. In the next section, we'll examine more usages of expressions for dealing with routes.

12.3 Routes

Routing is perhaps the biggest innovation of the ASP.NET MVC project—so big, in fact, it was included in the .NET Framework 3.5 SP1 release, well ahead of the ASP.NET MVC release. Like any new tool, routing is easy to abuse. Unless routes are tested thoroughly, changes to routes can break existing URLs. When URLs become public, changing them can break links, bookmarks, lower search rankings, and anger end users. Designing of custom routes and URL patterns should come from actual business requirements. In this section, we'll examine some common sense practices for routes, as well as some practices to ensure we don't break our application in the process.

12.3.1 Testing routes

When we do need custom routes, we need to ensure both that the routes we are creating are correct, and any existing routes are not modified. We can start off with the built-in routes, and lock those down with tests. The default route is shown in listing 12.26.

Listing 12.26 Default route in a new application

```
routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

routes.MapRoute(
    "Default",
    "{controller}/{action}/{id}",
    new { controller = "Home", action = "Index", id = "" }
);
```

For many applications, this route is sufficient and does not necessarily need to be tested on its own. If we added additional routing behavior, we would want to ensure that existing routes that follow this format are not broken. Before we start writing tests, we need to think of a few scenarios. The following URLs should work in the default sample application:

- /—maps to `HomeController.Index()`
- /home—maps to `HomeController.Index()`
- /home/about—maps to `HomeController.About()`

To make things more interesting, we'll add a simple `ProductController` to list, view, and search products, as shown in listing 12.27.

Listing 12.27 Simplified product controller

```
public class ProductController : Controller
{
    public ViewResult Index()
    {
        var products = new[]
        {
            new Product {Name = "DVD Player"},
            new Product {Name = "VCR"},
            new Product {Name = "Laserdisc Player"}
        };
        return View(products);
    }

    public ViewResult Show(int id)
    {
        return View(new Product {Name = "Hand towels"});
    }

    public ViewResult Search(string name)
    {
        return View("Show", new Product {Name = name});
    }
}
```

With our new controller, we want to support more interesting URL scenarios:

- /product/show/5—maps to `ProductController.Show`
- /product/SomeProductName—maps to `ProductController.Search(SomeProductName)`

Out of the box, the built-in routes support the first scenario, but not the second. Before we start messing around with our routes, we need to add tests to our existing scenarios. Testing routes is possible, but much easier with the testing extensions of the open source project, MvcContrib. We'll test the first scenario, as shown in listing 12.28.

Listing 12.28 Testing a blank URL

```
[Test]
public void Should_map_blank_url_to_home()
```

```
{
    "~/".Route().ShouldMapTo<HomeController>(c => c.Index());
}
```

Using extension methods, first transforms a string into a `Route` object with the `Route` extension method. Next, we use the `ShouldMapTo` extension method to assert that a route maps to the `Index` method on `HomeController`. `ShouldMapTo` is a generic method, taking an expression. It is similar to other expression-based methods such as `Html.ActionLink`. The expression is used to perform strongly typed reflection, as opposed to doing something like passing the controller and action name in as strings, which will fail under refactoring scenarios. Unfortunately, this test does not pass yet, as we have not called anything to set up our routes. We'll accomplish this in a test setup method to be executed before every test, as shown in listing 12.29.

Listing 12.29 Registering the routes in a setup method

```
[TestFixtureSetUp]
public void Setup()
{
    MvcApplication.RegisterRoutes(RouteTable.Routes);
}
```

With our setup in place, our test now passes. The next scenarios we want to test are the other built-in scenarios, as shown in listing 12.30.

Listing 12.30 Testing the built-in routing scenarios

```
[Test]
public void Should_map_home_url_to_home_with_default_action()
{
    "~/home".Route().ShouldMapTo<HomeController>(c => c.Index());
}

[Test]
public void Should_map_home_about_url_to_home_matching_method_name()
{
    "~/home/about".Route().ShouldMapTo<HomeController>(c => c.About());
}

[Test]
public void
     Should_map_product_show_with_id_to_product_controller_with_parameter()
{
    "~/product/show/5".Route().ShouldMapTo<ProductController>(
        c => c.Show(5));
}
```

With default scenarios added, we can now proceed with modifying our route to support the special case of a search term directly in the URL. Before we get there, let's make sure our routes don't already support this scenario by adding a test to verify the functionality. After all, if this test passes, our work is done! The new test is shown in listing 12.31.

Listing 12.31 New scenario routing product search terms

```
[Test]
public void Should_map_product_search_to_product_controller_with_parameter()
{
    "~/product/SomeProductName"
        .Route()
        .ShouldMapTo<ProductController>(c => c.Search("SomeProductName"));
}
```

Alas, our test fails, and our work is not yet done. The test fails with the message "MvcContrib.TestHelper.AssertionException : Expected Search but was SomeProductName." To make our test pass, we need to add the appropriate changes to the routes, as shown in listing 12.32.

Listing 12.32 Additional route for searching products

```
routes.MapRoute(
    "SearchProduct",
    "product/{name}",
    new { controller = "Product", action = "Search" }
);
```

With this addition to our routes, our new test passes, along with all the other tests. We were able to add a new route to our routing configuration with the assurance that we would not break the other URLs. Since URLs are now generated through routes in an MVC application, testing our routes becomes of utmost importance. The test helpers in MvcContrib wrapped all the ugliness that usually comes along with testing routes. In the next section, we'll examine action names and custom routes.

12.3.2 Action naming

Although the default routes in an MVC application match a URL to a method name on a controller, the defaults can be changed. As shown in section 12.3.1, we can map the second URL segment to a parameter on a specific action. When using the MVC extension points of the `ActionNameSelectorAttribute` and `ActionMethodSelectorAttribute`, the name of an action method on a controller does not exactly match the method name. The two concepts of *action name* and *action method* name are completely separate, and can be configured independently. We can configure an action method as shown in listing 12.33 to modify the action name.

Listing 12.33 Modifying the action name for an action method

```
public class ChangedActionNameController : Controller
{
    [ActionName("Foo")]
    public ActionResult Index()
    {
        return View();
    }
}
```

In the controller shown in listing 12.33, we specified that the action method name should be different from the action name. The action name, (originally the action method name, `"Index"`), is now `"Foo"`. Navigating to /changedactionnname or /changedactionname/index now results in a 404 Not Found error. The action name is now `"Foo"`, and we can only access this action through /changedactoinname/foo. As view names correspond to action names, not action method names, our view is named `"Foo.aspx"`.

But in most applications, we are better served adhering to the convention that action names match action method names. If method names differ from action names, we can no longer use expression-based URL generators. Our URL generation is now susceptible to subtle refactoring and renaming errors. This can be alleviated by introducing global constants for action names, but it still creates a string-based system, with another level of indirection between action methods and action names, that is not needed in many cases.

Consistency in action naming can reduce the complexity in your system. If your system is generally resource-based; that is, controllers are designed around individual entities (a `ProductController` and a `UserController`), RESTful-style action names introduce both discoverability on the client side, and predictable design on the developer side. Given a controller designed for products, the user interactions we might want to support include

- Listing products
- Showing one product
- Creating a new product
- Editing an existing product
- Deleting a product

Translated into controller actions, these would map to

- Index
- Show
- New
- Edit
- Delete

Because MVC action methods can be configured to accept only certain HTTP verbs, such as POST, we can design our controller with a set of overloaded action methods, one for viewing a form and one for receiving the posted form. If we had some sort of `Widget` resource in our system, our `WidgetsController` would look similar to listing 12.34.

Listing 12.34 A RESTful-style controller for managing `Widget` resources

```
public class WidgetsController : Controller
{
    public ActionResult Index()
    {
```

```
        return View();
    }

    public ActionResult Show(int id)
    {
        return View();
    }

    public ActionResult New()          ❶
    {
        return View();
    }

    [AcceptVerbs(HttpVerbs.Post)]                    ❷
    public ActionResult New(WidgetForm widget)
    {
        return RedirectToAction("Index");
    }

    public ActionResult Edit(int id)
    {
        return View();
    }

    [AcceptVerbs(HttpVerbs.Post)]
    public ActionResult Edit(WidgetForm widget)
    {
        return RedirectToAction("Index");
    }
}
```

Actions that support GET and POST verbs, the `New` and `Edit` actions, can be overloaded so that one method responds to GET requests ❶ and the other responds to POST requests ❷. To be clear, these actions don't match the definition of REST. However, the design would be simpler if all of the controllers dealing with a resource looked the same. If we went to a real RESTful architecture, using MvcContrib's `SimplyRestfulRouteHandler`, we could support all of the standard REST actions and corresponding HTTP verbs. Regardless of whether we want to adopt REST, having every action that shows a single entity or resource called `Show` makes new features easier to learn and makes the application easier to maintain.

12.4 Testing

The separation of concerns that the MVC pattern provides significantly increases testability for .NET web applications. Because controllers are normal classes, and actions are merely methods, we can load and execute actions and then examine the results. Even though testing controllers is simple, we must consider an important caveat. When we test a controller action, we are only able to write assertions for the behavior we can observe. The true test of a working application is running it in a browser, and there are significant differences between viewing a page in a browser and asserting results in a controller action test. We can assert that the correct view is chosen, but we cannot assert that the correct view is shown at runtime. We can assert that we put correct information into `ViewData`, but we cannot ensure that the view uses all of the

information we give it. We also cannot assert that all possible controller code paths place the necessary objects into ViewData. With action filters, it is quite possible that a view will need data that is not present. Controller action tests don't run the entire MVC engine, so things like ActionFilters are not executed. Although action unit tests add value, they don't replace end-to-end application-level testing. Before we examine the last mile of testing in UI tests, let's see how we can lock down the behavior in the rest of our MVC application through unit testing.

12.4.1 Controller unit tests

For controllers to be maintainable, they should be as light and skinny as possible, delegating all real domain work to other services. Our controller tests will reflect this choice, as assertions will be small and target only the following:

- What ActionResult was chosen
- What information was passed to the view, in ViewData or TempData

All other web-related information, whether it is security, cookies, or session variables, should be encapsulated in a domain-specific and domain-relevant interface. Although it eases testing, encapsulation and separation of concerns are the most significant reasons to leave these other HttpContext-related items out of controllers. The simplest example of a controller action is one that simply passes data into a view, as shown in listing 12.35.

Listing 12.35 A simple action

```
public ViewResult Index()
{
    var products = _productRepository.FindAll();

    return View(products);
}
```

In this example, productRepository is a private field of type IProductRepository, as shown in listing 12.36.

Listing 12.36 The controller with its dependency

```
public class ProductsController : Controller
{
    private readonly IProductRepository _productRepository;

    public ProductsController(IProductRepository productRepository)
    {
        _productRepository = productRepository;
    }
```

When we test the ProductsController, we don't need to supply the actual implementation of the IProductRepository interface. For the purposes of a unit test, we are testing only the ProductsController and no external dependency used. To maximize the localization of defects, our unit tests should test only a single class. We don't want a controller unit test to fail because we have a problem with our local database. In a

unit test, we'll have to pass a test double into the `ProductsController` repository. A test double is a stand-in for an actual implementation, but one that we can manipulate to force our class under test to execute specific code paths. Our controller unit test will need to set up the stubbed `IProductRepository` with dummy data, and then assert that the right action result is used, the right view is chosen, and the right data is passed to the view, as shown in listing 12.37.

Listing 12.37 Testing our `Index` action

```
[Test]
public void Index_should_use_default_view_and_repository_data()
{
    var products = new[]                                    ❶
    {
        new Product {Name = "Keyboard"},
        new Product {Name = "Mouse"}
    };

    var repository = Stub<IProductRepository>();            ❷
    repository.Stub(rep => rep.FindAll()).Return(products);  ❸

    var productsController = new ProductsController(repository);  ❹

    ViewResult result = productsController.Index();         ❺

    result.ViewName.ShouldEqual(string.Empty);              ❻
    result.ViewData.Model.ShouldEqual(products);            ❼
}
```

We set up product data for our test ❶. The values inside don't matter for the purposes of our unit test, but aid in debugging if our test fails for an unknown reason. We create a stub of our `IProductRepository` by calling a method on our base test class ❷. This method is a wrapper around Rhino Mocks, a popular test double creation and configuration framework. After we create a test double of our `IProductRepository`, we stub out the call to `FindAll` to return our array of `Products` we created earlier ❸. With the stubbed `IProductRepository`, we create a `ProductsController` ❹.

With all of the classes and test doubles set up for our unit test, we can execute our controller action and capture the resulting `ViewResult` object ❺. We assert that the `ViewName` should be an empty string ❻ (signifying we use the Index view), and that the model passed to the view is our original array of products ❼. Our test passes with the implementation of our action shown in listing 12.35.

A two-line action method is tested easily, but is not very interesting. In a more interesting scenario, we edit a model, then post it to a form. We expect several things to happen:

- Check the model state for errors
- If errors exist, show the original view
- If not, save the model and redirect back to the index

Let's start with the error path, where a user entered incorrect information. We'll assume that model state errors are populated through other means as a result of validation,

perhaps through a model binder or action filter. For the purposes of our test, shown in listing 12.38, the means of validation is not important, but rather, how the controller behaves under this condition.

Listing 12.38 Testing the edit action when errors are present

```
[Test]
public void Edit_should_redirect_back_when_model_errors_present()
{
    var badProduct = new Product { Name = "Bad value" };

    var repository = Stub<IProductRepository>();

    var productsController = new ProductsController(repository);
    productsController
        .ModelState.AddModelError("Name", "Name already exists");     ❶

    var result = productsController.Edit(badProduct);                 ❷

    result.AssertViewRendered().ViewName.ShouldEqual(string.Empty);   ❸
    repository.AssertWasNotCalled(rep => rep.Save(badProduct));       ❹
}
```

To force our controller into an invalid model state, we need to add a model error to `ModelState` with the `AddModelError` method ❶. After setting up our controller, we invoke the `Edit` action ❷, and examine the result returned. We assert that a view is rendered with the `AssertViewRendered` method ❸, which returns a `ViewResult` object. The `ViewName` on the `ViewResult` should be an empty string, signifying the `Edit` view is rerendered. Finally, we assert that the `Save` method on our repository was not called ❹. This negative assertion ensures we don't try to save our `Product` if it has validation problems. Normally, we would create a separate presentation model specifically for the form, but in this example, we use our domain model directly. We tested the error condition, and now we need to test our controller in the positive condition that our model didn't have any validation problems, as shown in listing 12.39.

Listing 12.39 Testing our controller action when no errors are present

```
[Test]
public void Edit_should_save_and_redirect_when_no_model_errors_present()
{
    var goodProduct = new Product { Name = "Good value" };

    var repository = Stub<IProductRepository>();

    var productsController = new ProductsController(repository);

    var result = productsController.Edit(goodProduct);     ❶

    result
        .AssertActionRedirect()                            ❷
        .ToAction<ProductsController>(c => c.Index());     ❸

    repository.AssertWasCalled(rep => rep.Save(goodProduct));
}
```

In this test, we set up our dummy product and controller in a manner similar to the last test, except this time we don't add any model errors to our `ModelState`. We invoke the `Edit` action with the product we created ❶, and then verify values on the result. We use the MvcContrib project's `AssertActionRedirect` ❷ to assert that the result of our action redirects to another action, specifically to the `Index` action. The `ToAction` method allows us to assert that we redirect to a specific action using a strongly typed expression ❸. Because we use expressions here, our test won't break if we rename the `Index` action method name. To make both of these tests pass, our action looks like listing 12.40.

Listing 12.40 Implementation of the `Edit` action

```
[AcceptPost]
public ActionResult Edit(Product product)
{
    if (!ModelState.IsValid)      ❶
    {
        return View(product);     ❷
    }
    _productRepository.Save(product);

    return this.RedirectToAction(c => c.Index());     ❸
}
```

In our `Edit` action, we check for any `ModelState` errors with the `IsValid` property ❶, and return a `ViewResult` with our original `Product` ❷. Our `Edit` view likely will use styling to highlight individual model errors and display a validation error summary. If there are no validation errors, we save the `Product` and redirect back to the `Index` action ❸. With our controller's behavior locked down sufficiently, we can feel confident we can modify our `Edit` action in the future and know if our change breaks existing functionality. In the next section, we'll examine strategies for testing custom model binders.

12.4.2 Model binder unit tests

Custom model binders eliminate much of the boring plumbing that often clutters action methods with code not pertinent to the true purpose of the action method. But with this powerful tool comes the need for thorough testing. Our infrastructure needs to be rock solid, as it can execute on a large majority of requests. Testing model binders is not as straightforward as testing action methods, but it is possible. The amount of testing needed varies depending on what you are doing with your custom model binder. Simply implementing the `IModelBinder` interface likely means you'll only need to worry about one single `BindModel` method and only a `ModelBindingContext` during testing. Inheriting from `DefaultModelBinder` is a bit more challenging, as any code we add will execute alongside other code that we don't own. We must ensure that any behavior we add works correctly in the context of the other responsibilities of the base `DefaultModelBinder` class. The `DefaultModelBinder` class design has extensibility in

mind, and key extension points are available through specific method overrides, but we still need to test these methods in the context of an entire binding operation (such as a single `BindModel` call).

In section 12.1.3, we examined creating a custom model that bound entities from a repository, as shown in listing 12.41.

Listing 12.41 Entity model binder implementation

```
public object BindModel (
    ControllerContext controllerContext,
    ModelBindingContext bindingContext)
{
    ValueProviderResult value =
        bindingContext.ValueProvider [bindingContext.ModelName];

    if (value == null)
        return null;

    if (string.IsNullOrEmpty(value.AttemptedValue))
        return null;

    var entityId = new Guid(value.AttemptedValue);

    Type repositoryType = typeof(IRepository<>)
         .MakeGenericType(bindingContext.ModelType);
    var repository = (IRepository)IoC.Resolve(repositoryType);

    PersistentObject entity = repository.GetById(entityId);

    return entity;
}
```

We didn't add any tests in our original example, so let's add some now. We have several guard clauses protecting against bad input. However, we didn't include the check for a user or part of our application puting an invalid GUID into the querystring (or form variable). Rather than allow an exception to be thrown during binding, we would like to handle this by returning null, as shown in the test in listing 12.42.

Listing 12.42 Test for bad GUID values

```
[Test]
public void Should_resolve_bind_to_null_when_guid_not_in_correct_format()
{
    var valueProviderDictionary = new ValueProviderDictionary(null)
    {
        {
           "ProductId",
           new ValueProviderResult ("NotAGuid", "NotAGuid", null)     (1)
        }
    };

    var bindingContext = new ModelBindingContext     (2)
    {
        ModelName = "ProductId",
        ValueProvider = valueProviderDictionary
```

```
    };

    var binder = new EntityModelBinder ();
    object model = binder.BindModel (null, bindingContext);

    model.ShouldBeNull();
}
```

Our model binder uses only a `ModelBindingContext`, not the `ControllerContext`. We need only focus on creating a `ModelBindingContext` representative of an invalid GUID value. First, we create a `ValueProviderDictionary`, with a single entry for a `ProductId` parameter ❶. For the raw and attempted values in the `ValueProviderResult`, we'll substitute bad GUID values, to force our model binder to throw an exception. With our `ValueProviderDictionary` assembled, we can create our `ModelBindingContext` ❷, using the same `ModelName` as was used in our `ValueProviderDictionary`. Because we use the `ModelName` directly to look up `ValueProviderResults` in our model binder, any mismatch will cause our custom model binder to not execute the code we are interested in. When we execute this unit test, it fails with a `System.FormatException`, because our model binder is not yet able to handle invalid GUIDs. To make our test pass, we can either parse the input string using regular expressions, or use a `try…catch` block. For simplicity, we'll use the exception handling method, with the additions shown in listing 12.43.

Listing 12.43 Modifying the GUID parsing code to handle invalid values

```
Guid entityId;

try
{
    entityId = new Guid(value.AttemptedValue);
}
catch (FormatException)     ❶
{
    return null;
}
```

With these changes, our test now passes. We surrounded our original GUID constructor with a `try..catch` block for the specific `FormatException` type thrown when the parsed value is not of the right format ❶. There are other interesting scenarios we could add tests for, but all of them employ the same technique of creating a `ModelBindingContext` representative of a certain model-binding scenario. Unit tests for model binders go quite a long way to proving the design of a model binder, but still don't guarantee a working application.

Model binders are one cog in a larger machine, and only through testing that larger part can we have complete confidence in our model binders. It can often take quite a bit of trial and error to get the model binder to function correctly. When it is working correctly, we need only to construct the context objects used by our model binder in our unit test to recreate those scenarios. Unfortunately, merely looking at a model binder may not show you how to construct the context objects it uses. A common test failure

is a NullReferenceException, where a call to an MVC framework method requires other supporting objects in place. The easiest way to determine what pieces your model binder needs in place is to simply write a test and see if it passes. If it does not pass because of an exception, keep fixing the exceptions, often by supplying test doubles, until your test passes or fails due to an assertion failure. In the next section, we'll examine testing action filters.

12.4.3 Action filter unit tests

The story for testing action filters is very similar to that for testing model binders. Unit testing is possible, and its difficulty is directly proportional to how much the filter relies on the context objects. Generally, the deeper the filter digs in to the context object, the more we'll need to be set up or mocked in a unit test. Table 12.1 illustrates the types of filters and the context objects used for each.

Table 12.1 Filters and their supporting context objects

Filter type	Method	Context object
IActionFilter	OnActionExecuted	ActionExecutedContext
	OnActionExecuting	ActionExecutingContext
IAuthorizationFilter	OnAuthorization	AuthorizationContext
IExceptionFilter	OnException	ExceptionContext
IResultFilter	OnResultExecuted	ResultExecutedContext
	OnResultExecuting	ResultExecutingContext

Each context object has its own difficulties for testing, as each has its own dependencies for usage. All context objects have a no-argument constructor, and a unit test may be able to use the context object as is without needing to supply it with additional objects. Although your filter may use only one piece of the context object, you may find yourself needing to supply mock instances of more pieces, as many of the base context object constructors have null argument checking. You may find yourself far down a long path that leads to supplying the correct dependencies for a context object, and these dependencies may be several levels deep. Let's add tests to the filter shown in listing 12.44.

Listing 12.44 Simple action filter

```
public class CurrentUserFilter : IActionFilter
{
    private readonly IUserSession _session;

    public CurrentUserFilter (IUserSession session)
    {
        _session = session;
    }
```

```
    public void OnActionExecuting(ActionExecutingContext filterContext)
    {
        ControllerBase controller = filterContext.Controller;
        User user = _session.GetCurrentUser();
        if (user != null)
        {
            controller.ViewData.Add(user);
        }
    }

    public void OnActionExecuted(ActionExecutedContext filterContext)
    {
    }
}
```

In this filter, we have the requirement that a `User` object is needed for a component in the view, likely for displaying the current user in a widget. Our `CurrentUserFilter` depends on an `IUserSession`, whose implementation contains the logic for storing and retrieving the current logged in user from the session. Our filter retrieves the current user and places it into the controller's `ViewData`. The controller is supplied through the `ActionExecutingContext` object. If possible, during unit testing, we prefer to use the no-argument constructor and supply any additional pieces by merely setting the properties on the context object. The `ActionExecutingContext` type has setters for the `Controller` property, so we'll be able to use the no-argument constructor and not worry about the larger, parameter-full constructor. Our complete unit test, shown in listing 12.45, is able to create a stub implementation for only the parts used in our filter.

Listing 12.45 Action filter unit test

```
[TestFixture]
public class CurrentUserFilterTester : TestClassBase
{
    [Test]
    public void Should_pass_current_user_when_user_is_logged_in ()
    {
        var loggedInUser = new User();

        var userSession = Stub<IUserSession>();                    (1)
        userSession.Stub(session => session.GetCurrentUser())      (2)
            .Return(loggedInUser);

        var filterContext = new ActionExecutingContext             (3)
        {
            Controller = Stub<ControllerBase>()                    (4)
        };

        var currentUserFilter = new CurrentUserFilter (userSession);   (5)
        currentUserFilter.OnActionExecuting(filterContext);

        filterContext.Controller.ViewData                          (6)
            .Get<User>().ShouldEqual(loggedInUser);
    }
}
```

Our `CurrentUserFilter` depends on an implementation of an `IUserSession` interface ❶, which we supply using the `Stub` method. The `Stub` method comes from the `TestClassBase` class, and is a wrapper around Rhino Mocks' `CreateStub` method. Next, we stub the `GetCurrentUser` method on our stub `IUserSession` to return the `User` object created earlier ❷. Because the actual implementation of `IUserSession` requires the full `HttpContext` to be up and running, by supplying a fake implementation, we get much finer control over the inputs to our filter object.

Next, we create our `ActionExecutingContext` ❸, but call only the no-argument constructor. The controller can be any controller instance, and we again use Rhino Mocks to create a stub implementation of `ControllerBase` ❹. Rhino Mocks creates a subclass of `ControllerBase` at runtime, which saves us from using an existing or dummy controller class. In any case, the `ControllerBase` provides `ViewData`, so we don't need to provide any stub implementation for that property. With our assembled `ActionExecutingContext` and stubbed implementation of `IUserSession`, we can create and exercise our `CurrentUserFilter` ❺. The `OnExecutingMethod` does not return a value, so we need to examine only the `ActionExecutingContext` passed in. We assert that the controller's `ViewData` contains the same logged-in user created earlier ❻, and our test passes!

Getting to this point required trial and error to understand what the context object requires for execution. Because filters are integrated and specific to the MVC framework, it can be fruitless to try to write filters test-first, as only the fact that the complete website is up and running proves the filter is working properly. We supplied dummy implementations of the context objects, but constructed them in a way that the MVC framework will likely not use. In the next section, we'll examine how to automate tests with the entire website up and running through automated UI tests.

12.4.4 Testing the last mile with UI tests

In this chapter thus far, we examined testing individual components of ASP.NET MVC, including routes, controllers, filters, and model binders. Although unit testing each component in isolation is important, the final test of a working application is interaction with a browser against a live instance. With all of the components that make up a single request, whose interaction and dependencies can become complex, it is only through browser testing that we can ensure our application works as desired from end to end. While developing an application, we often launch a browser to manually check that our changes are correct and produce the intended behavior.

In many organizations, manual testing is formalized into a regression testing script to be executed by development or QA personnel before a launch. Manual testing is slow and quite limited, as it can take minutes to execute a single test. In a large application, regression testing is minimal at best and woefully inadequate in most situations. Fortunately, many free automated UI testing tools exist. Some of the more popular tools are listed here:

- WatiN (http://watin.sourceforge.net/)
- Watir (http://wtr.rubyforge.org/)
- Selenium (http://seleniumhq.org/)
- QUnit (http://docs.jquery.com/QUnit)—for testing JavaScript

In addition to these open source projects, many commercial products on the market provide additional functionality or integration with bug reporting systems or work item tracking systems, such as Microsoft's Team Foundation Server. However, the tools are not tied to any testing framework, so integration with an existing project is rather trivial.

In this section, we'll examine UI testing with WatiN, which provides easy integration with unit testing frameworks. WatiN, an acronym of *web application testing in .NET*, is a .NET library that provides an interactive browser API to both interact with the browser, by clicking links and buttons for example, as well as find elements in the DOM.

Testing with WatiN usually involves interacting with the application to submit a form, then checking the results in a view screen. Because WatiN is not tied to any specific unit testing framework, we can use any unit testing framework we like. The testing automation platform Gallio (http://www.gallio.org/) provides important additions that make automating UI tests easier:

- Test steps for logging individual interactions in a single test
- Running tests in parallel
- Ability to embed screenshots in the test report (for failures)

To get started, we need to download and install Gallio. Gallio includes an external test runner (Icarus), as well as integration with many unit testing runners, including TestDriven.NET, ReSharper, and others. Also included in Gallio is MbUnit, a unit testing framework which we'll use to author our tests. With Gallio downloaded and installed, we need to create a Class Library project and add references to both Gallio.dll and MbUnit.dll. Next, we need to download WatiN and add a reference in our test project to the WatiN.Core.dll assembly. With our project references done, we are ready to create a simple test. One of the most basic, but useful scenarios in our application is to test to see if we can log in to our application. Testing manually, this would mean

1. Navigating to the login URL
2. Entering username and password
3. Clicking the Log in button
4. Checking that the login widget at the top of the screen has the correct name

Because we'll want common functionality and configuration in all of our test classes that use WatiN, we'll create a base test class, as shown in listing 12.46.

Listing 12.46 Web test base class

```
[TestFixture]
[ApartmentState (ApartmentState.STA)]
```

```
public class WebTestBase
{
}
```

The first attribute on our `WebTestBase` class should be familiar; it is the MbUnit attribute for tagging a class as a `TestFixture`. The next attribute is not as well known. Because WatiN uses COM to communicate with Internet Explorer (IE), and the COM IE wrapper is not thread-safe, we must configure our unit test runner to use a single-threaded apartment (STA). Each unit test runner is configured differently and in MbUnit's case, we use the `ApartmentStateAttribute` with and `ApartmentState` value of `ApartmentState.STA`. With this attribute applied to our `WebTestBase` class, we need to configure this setting only once in our test project, as long as all of our tests use `WebTestBase` as a base class.

Next, we can create a new test that performs the steps listed earlier in this section, as shown in listing 12.47.

Listing 12.47 Testing the login screen

```
public class LoginScreen : WebTestBase
{
    [Test]
    public void Can_log_in_successfully()
    {
        using (var ie = new IE ("http://localhost:8082/Login"))     ❶
        {
            ie.TextField (Find.ByName ("Username")).TypeText("admin");     ❷
            ie.TextField (Find.ByName ("Password")).TypeText("password");

            ie.Button(Find.ByName ("login")).Click();

            Assert.IsTrue(ie.ContainsText ("Joe User"));
        }
    }
}
```

Our `LoginScreen` class inherits from the `WebTestBase` class created earlier. In the `LoginScreen` test class, we define one test, "Can_log_in_successfully." Inside this test, we first create a new instance of the WatiN `IE` object ❶. The `IE` class has a constructor that takes a URL as a parameter, which causes the IE browser to immediately launch at the specified URL. We hardcoded the correct starting URL so that the IE browser immediately navigates to the login screen. If the starting URL needs to be configured, we could pull this information from a configuration file. The lifetime of the `IE` object is wrapped in a using statement block, to ensure that our COM resources are disposed of properly.

The `IE` object is our primary source of interaction with the browser. It includes a variety of methods to locate elements in the DOM, as well as methods to interact with the browser's periphery, such as cookies, dialog boxes, and so on. Our interaction will deal mainly with locating and manipulating DOM elements, but other browser interaction is available if needed. Back in our test, the next two lines use the `TextField`

method ❷ to locate the HTML INPUT elements of type TEXT. The `TextField` method takes a variety of arguments, each enabling a different way to search for elements. With ASP.NET MVC, we can use the `Constraint` overload, and use the `Find` static class to build a `Constraint` object to match the element we need. Other options include a string for an element ID, a regular expression, or a custom callback function. For our purposes, we'll stick mainly with the `Find.ByName` constraint. With ASP.NET Web Forms, it was more common to use regular expressions, as element ID and names were not entirely deterministic. The MVC framework gives us complete control over element IDs and names.

The `TextField` method returns a single `TextField` object. We use the `TypeText` method to fill in text into both the username and password fields. In this test, we didn't set up any login information beforehand, and we know that this login information will work for a clean build of CodeCampServer. Typically, we'll set up all entities needed for a test in a setup method. After filling in the username and password, we use the `Button` method in combination with the `Find.ByName` constraint to locate the login button and click it with the `Click` method. If our login is correct, we'll be redirected in the browser to the home page, and our user's name will appear at the top. To verify this, we use the `ContainsText` with our user's name and assert that our user's name is found.

With our basic test in place, we can execute this test in the Gallio Icarus test runner, shown in figure 12.3.

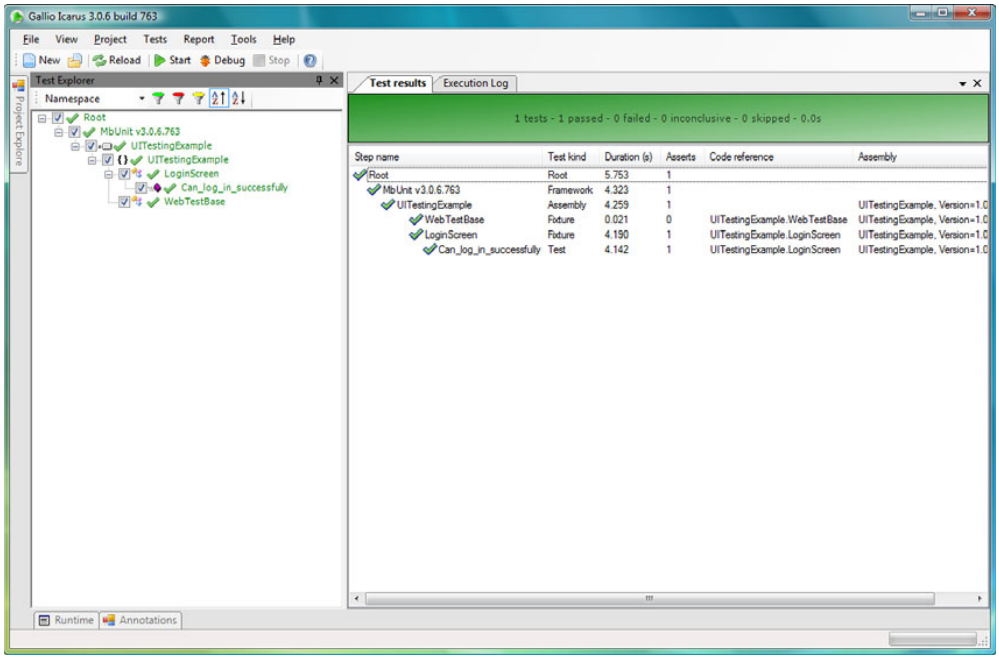


Figure 12.3 Simple passing login screen test

In our test, we referenced all of the input elements by name, but how did we know what name to look for? In older browsers, this meant viewing the HTML source. In modern browsers, including IE8 and Chrome, a built-in HTML inspector picks HTML elements by clicking them to bring the specific HTML element into a readable interface. Google's Chrome HTML inspector, shown in figure 12.4, allows us to click an element on the browser to determine relevant information, such as element names.

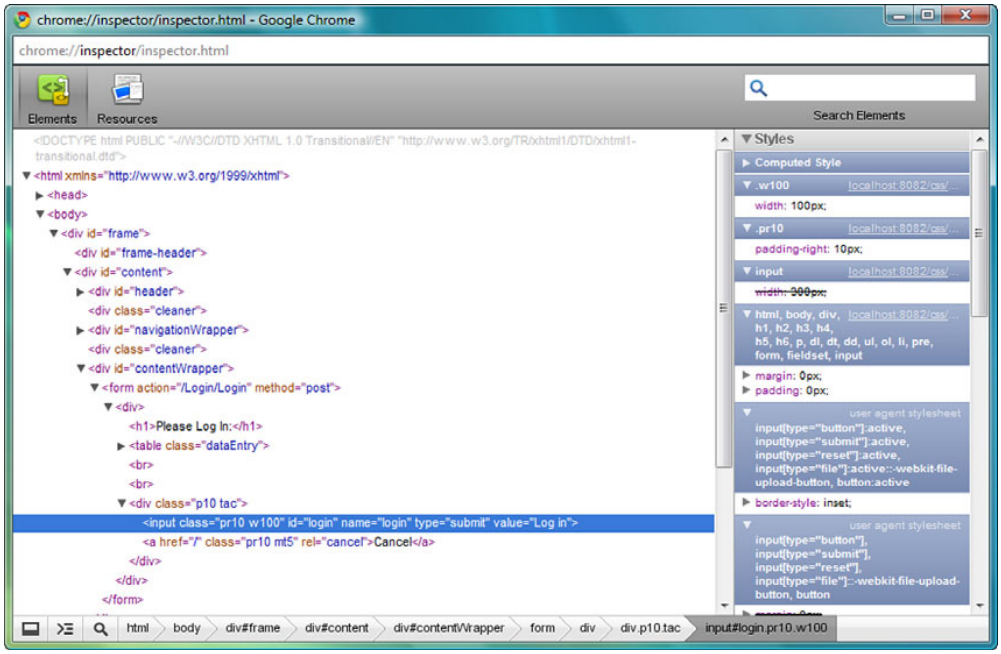


Figure 12.4 Google Chrome HTML inspector with our login button highlighted

Other browsers have extensions for this purpose, including Firebug (http://getfirebug.com/) for Firefox, and the IE Web Developer Toolbar (http://www.microsoft.com/downloads/details.aspx?familyid=E59C3964-672D-4511-BB3E-2D5E1DB91038) for versions previous to IE8. With these tools, we need only to click elements on a live browser and inspect their element names or IDs for our UI tests. In figure 12.4, we have the login button from our login screen selected, leaving guesswork or hunting through our project behind.

But what if our test fails? One of the features of MbUnit is the ability to embed images into test reports, and one of the features of WatiN is to capture images. First, we'll create a failing test in as shown in listing 12.48.

Listing 12.48 Intentional test failure

```
[Test]
public void Intentional_failure()
```

```
{
    using (var ie = new IE ("http://localhost:8082/Login"))
    {
        ie.TextField (Find.ByName ("Username")).TypeText("admin");
        ie.TextField (Find.ByName ("Password")).TypeText("password");

        ie.Button(Find.ByName ("login")).Click();

        Assert.IsTrue(ie.ContainsText ("Joe Schmoe"));
    }
}
```

For this intentionally failing test, we change the text of the name asserted to an incorrect name, "Joe Schmoe." Running this test proves our failure, but we would like to capture the screenshot as part of the failure. Because we created the `WebTestBase` class earlier, we can centralize all failure behavior in one place. We can create a teardown method, run after every test, and check to see if there were any failures in our test. If so, we take a screenshot using WatiN and embed the image into Gallio's test results. To accomplish all of this, we'll need to make more modifications to our `WebTestBase` class, as taking a screenshot requires the original instance of the `IE` object. Because our original test had the `IE` object in a `using` block, it won't be available to our teardown method without modifications to our test. Instead of instantiating our IE object in each test, we'll do so in our `WebTestBase` in a `SetUp` method, as shown in listing 12.49.

Listing 12.49 Modified `WebTestBase` setting up the `IE` object

```
protected IE Browser { get; private set; }

[SetUp]
public void SetUp()
{
    Browser = new IE ("http://localhost:8082/Login");
}
```

Before each test executes, we create an `IE` instance and assign it to our protected `Browser` property. Our original failing test now needs to use the `Browser` property instead of creating the `IE` object itself, as shown in listing 12.50.

Listing 12.50 Modifying the failing test to use the `Browser` property

```
[Test]
public void Intentional_failure()
{
    Browser.TextField (Find.ByName ("Username")).TypeText("admin");
    Browser.TextField (Find.ByName ("Password")).TypeText("password");

    Browser.Button(Find.ByName ("login")).Click();

    Assert.IsTrue(Browser.ContainsText ("Joe Schmoe"));
}
```

With our `IE` object now managed by our base test class, we can introduce a `TearDown` method to check for test failures and capture screenshots. Even if we didn't include

the screenshot concept, we still need to add code in a teardown method to dispose of our IE instance properly. Our TearDown method is shown in listing 12.51.

Listing 12.51 Teardown method with image capturing and logging

```
[TearDown]
public void TearDown()
{
    try
    {
        if (TestContext.CurrentContext
                .Outcome.Status == TestStatus.Failed)     ❶
        {
            var writer = TestLog.Writer.Default;                    ❷
            using (writer.BeginSection("Test failed on this page"))    ❸
            {
                writer.Write("Url: ");
                using (writer.BeginMarker(Marker.Link(Browser.Url)))    ❹
                {
                    writer.WriteLine(Browser.Url);
                }
                var imageCapturer = new CaptureWebPage (Browser);    ❺
                var image = imageCapturer                            ❻
                    .CaptureWebPageImage(false, false, 100);
                writer.EmbedImage("Failure.png", image);
            }
        }
    }
    finally
    {
        Browser.Close();     ❼
        Browser = null;
    }
}
```

In a try-finally block, we separate the image capturing and logging from managing the IE instance. The IE browser should always be discarded at teardown, regardless of whether an exception happens during image capturing ❼. The try-finally block ensures our IE instance is disposed of properly. Inside the try block, we first check Gallio's test status in the TestContext object ❶. We only want to capture screenshots in the event of a failing test. Next, we create a reference to the default log writer for Gallio ❷. Gallio supports multiple nested log streams for complex test reports, but in our case, the default will suffice.

To create sections in our log output, we use the BeginSection method ❸. We might have more sections logged detailing the steps executed in our test, so a separate section for the error helps distinguish it in the final report. We also write the original URL of the screen with the error for informational purposes. Using the Marker.Link method ❹ generates a clickable link in the final report, helpful to quickly traverse to the failing screen. We are ready to capture the image.

We create a CaptureWebPage object ❺, passing in the IE instance stored in our test class. Next, we create an Image object and capture a screenshot using the

`CaptureWebPageImage` method. We use the `EmbedImage` method ❻ on our log writer object, providing the image object and a file name. Running this test in our Icarus test runner gives us a nice screenshot of our failure, as shown in figure 12.5.

Gallio is a powerful tool for creating UI tests when combined with WatiN. We can create a wrapper over the WatiN browser calls, which can be difficult to read, as well as more fluent calls that take advantage of strongly typed views, expressions, and Gallio's test steps. With test steps and a simple wrapper, we can log all interaction with IE as a sort of test script, so that we can easily read exactly what our test performed, and exactly where it failed in the context of a user's actions, instead of a stack trace. We might rather know that a test failed when the user clicked Submit Order, rather than receive a line number in a file. With Gallio and WatiN, this is possible. Tests that might take weeks to execute manually can finish in an hour.

UI tests are much, much slower than unit and integration tests, but they are vital in ensuring our application works end to end. Because of the speed of these tests, their use should be reserved for scenario-based tests, happy-path or black-box testing, and regression tests. Unless care is taken to ensure strongly typed tests and to avoid the magic strings we examined earlier, UI tests become quite brittle. It's worth noting that most applications are not easily testable without modifications. Just as we have to design our code for testability, we need to design our UI for testability. This might include putting IDs or special class names around certain data-driven elements, or sharing the view types with our UI tests to ensure that the exact same HTML element names are used for both HTML generation and UI testing. These changes don't affect

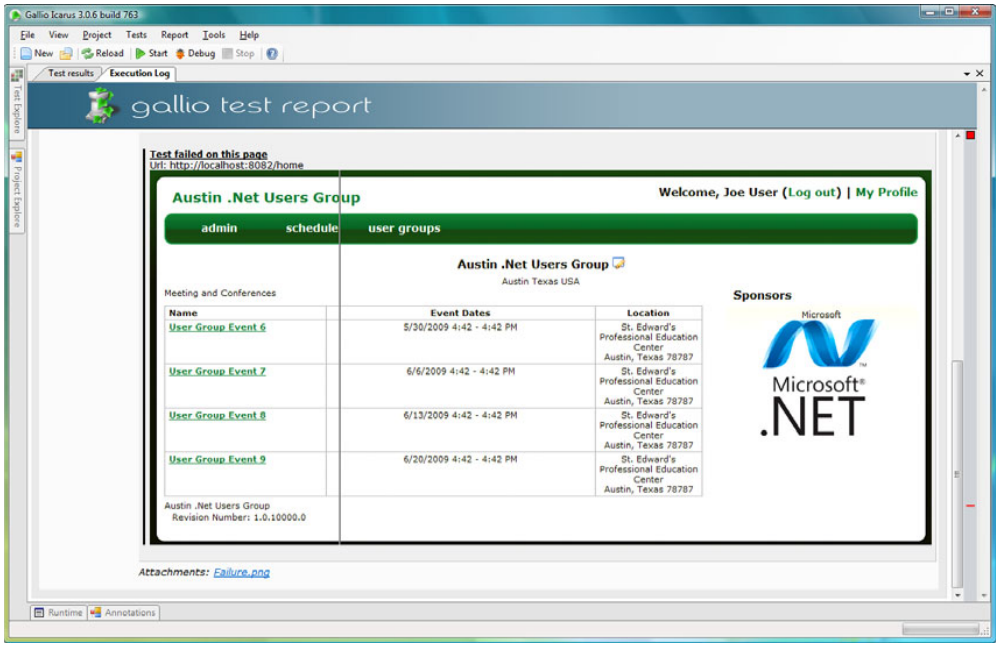


Figure 12.5 Our test report including a screenshot

the end-user experience, and allow us as developers to focus on adding value, rather than fixing brittle tests.

12.5 Summary

In this chapter, we explored many of the extension points and major feature areas of ASP.NET MVC and discovered how best to take advantage of these areas in a maintainable manner. Although not every practice applies in every context, it is important to consider all the options available, and the benefits and tradeoffs of each before proceeding with a design. If you go down a path with filters and magic strings in `ViewData`, you might not like the end result. Instead, we can consider the long-term viability of each option and choose the most appropriate path for each situation. Some practices are strongly recommended for a maintainable and easily testable codebase, such as strongly typed views. Others, such as convention-based, REST-style action names are appropriate only in resource-centric applications.

Duplication is one of the biggest causes of development attrition, whether using ASP.NET MVC or another framework. The techniques used to remove duplication have changed from classic Web Forms, from custom model binders, to action filters and partials in our views. Although each of these extension points is powerful, none is appropriate in every context. We examined many of the options for eliminating duplication in our controllers and views, as well as elaborating on the right contexts for each of these options.

We focused on testing these extension points. Because these extension points can be executed on every request, it is vital to ensure that these extension points behave as desired. However, the true test of a working MVC application is using it in a browser. We finished our testing discussion by examining UI testing with WatiN and Gallio, taking advantage of features in both products to capture screenshots from failures and logging meaningful test messages. In the next chapter, we'll examine a variety of real-world scenarios in the form of in-depth recipes.

ASP.NET/WEB DEVELOPMENT

ASP.NET MVC IN ACTION

Jeffrey Palermo • Ben Scheirman • Jimmy Bogard
FOREWORD BY Phil Haack

Free eBook
SEE INSERT

ASP.NET MVC implements the Model-View-Controller pattern on the ASP.NET runtime. It works well with open source projects like NHibernate, Castle, StructureMap, AutoMapper, and MvcContrib.

ASP.NET MVC in Action is a guide to pragmatic MVC-based web development. After a thorough overview, it dives into issues of architecture and maintainability. The book assumes basic knowledge of ASP.NET (v. 3.5) and expands your expertise. Some of the topics covered:

- How to effectively perform unit and full-system tests.
- How to implement dependency injection using StructureMap or Windsor.
- How to work with the domain and presentation models.
- How to work with persistence layers like NHibernate.

The book's many examples are in C#.

Jeffrey Palermo is co-creator of MvcContrib. **Jimmy Bogard** and **Ben Scheirman** are consultants and .NET community leaders. All are Microsoft MVPs and members of ASPInsiders.

For online access to the authors and a free ebook for owners of this book, go to manning.com/ASP.NETMVCinAction

"Shows how to put all the features of ASP.NET MVC together to build a great application."
—From the Foreword by Phil Haack
Senior Program Manager
ASP.NET MVC Team, Microsoft

"This book put me in control of ASP.NET MVC."
—Mark Monster
Software Engineer, Rubicon

"Of all the offerings, this one got it right!"
—Andrew Siemer
Principal Architect, OTX Research

"Highly recommended for those switching from Web Forms to MVC."
—Frank Wang, Chief Software Architect, DigitalVelocity LLC

ISBN 13: 978-1-933988-62-7
ISBN 10: 1-933988-62-2
54499
9 781933 988627

MANNING $44.99 / Can $62.99 [INCLUDING eBOOK]