CHAPTER 3

The MVC Pattern

Before I start digging into the details of the ASP.NET MVC Framework, I want to make sure you are familiar with the MVC design pattern and the thinking behind it. In this chapter, I describe the following:

- The MVC architecture pattern
- Domain models and repositories
- Creating loosely coupled systems using dependency injection (DI)
- The basics of automated testing

You might be familiar with some of the ideas and conventions I discuss in this chapter, especially if you have done advanced ASP.NET or C# development. If not, I encourage you to read carefully—a good understanding of what lies behind MVC can help put the features of the framework into context as you continue through the book.

The History of MVC

The term *model-view-controller* has been in use since the late 1970s and arose from the Smalltalk project at Xerox PARC, where it was conceived as a way to organize some early GUI applications. Some of the fine detail of the original MVC pattern was tied to Smalltalk-specific concepts, such as *screens* and *tools*, but the broader concepts are still applicable to applications—and they are especially well suited to Web applications.

Interactions with an MVC application follow a natural cycle of user actions and view updates, where the view is assumed to be stateless. This fits nicely with the HTTP requests and responses that underpin a Web application.

Further, MVC forces a *separation of concerns*—the domain model and controller logic are decoupled from the user interface. In a Web application, this means that the HTML is kept apart from the rest of the application, which makes maintenance and testing simpler and easier. It was Ruby on Rails that led to renewed mainstream interest in MVC and it remains the implementation template for the MVC pattern. Many other MVC frameworks have since emerged and demonstrated the benefits of MVC—including, of course, ASP.NET MVC.

Understanding the MVC Pattern

In high-level terms, the MVC pattern means that an MVC application will be split into at least three pieces:

- *Models*, which contain or represent the data that users work with. These can be simple *view models*, which just represent data being transferred between views and controllers; or they can be *domain models*, which contain the data in a business domain as well as the operations, transformations, and rules for manipulating that data.
- *Views*, which are used to render some part of the model as a user interface.
- *Controllers*, which process incoming requests, perform operations on the model, and select views to render to the user.

Models are the definition of the universe your application works in. In a banking application, for example, the model represents everything in the bank that the application supports, such as accounts, the general ledger, and credit limits for customers—as well as the operations that can be used to manipulate the data in the model, such as depositing funds and making withdrawals from the accounts. The model is also responsible for preserving the overall state and consistency of the data—for example, making sure that all transactions are added to the ledger, and that a client doesn't withdraw more money than he is entitled to or more money than the bank has.

Models are also defined by what they are *not* responsible for: models don't deal with rendering UIs or processing requests—those are the responsibilities of *views* and *controllers*. *Views* contain the logic required to display elements of the model to the user—and nothing more. They have no direct awareness of the model and do not directly communicate with the model in any way. *Controllers* are the bridge between views and the model—requests come in from the client and are serviced by the controller, which selects an appropriate view to show the user and, if required, an appropriate operation to perform on the model.

Each piece of the MVC architecture is well-defined and self-contained—this is referred to as the *separation of concerns*. The logic that manipulates the data in the model is contained *only* in the model; the logic that displays data is *only* in the view, and the code that handles user requests and input is contained *only* in the controller. With a clear division between each of the pieces, your application will be easier to maintain and extend over its lifetime, no matter how large it becomes.

Understanding the Domain Model

The most important part of an MVC application is the domain model. We create the model by identifying the real-world entities, operations, and rules that exist in the industry or activity that the application must support, known as the *domain*.

We then create a software representation of the domain: the *domain model*. For the purposes of the ASP.NET MVC Framework, the domain model is a set of C# types (classes, structs, etc.), collectively known as the *domain types*. The operations from the domain are represented by the methods defined in the domain types, and the domain rules are expressed in the logic inside of these methods—or, as you saw in the previous chapter, by applying C# attributes. When an instance of a domain type is created to represent a specific piece of data, it is called a *domain object*. Domain models are usually persistent and long-lived—there are lots of different ways of achieving this, but relational databases remain the most common choice.

In short, a domain model is the single, authoritative definition of the business data and processes within your application. A *persistent* domain model is also the authoritative definition of the state of your domain representation.

The domain model approach solves many of the problems that arise when maintaining an application. If you need to manipulate the data in your model or add a new process or rule, the domain model is the only part of your application that has to be changed.

■ **Tip** A common way of enforcing the separation of the domain model from the rest of an ASP.NET MVC application is to place the model in a separate C# assembly. In this way, you can create references *to* the domain model from other parts of the application but ensure that there are no references in the other direction. This is particularly useful in large-scale projects. I demonstrate this approach in the example I start building in Chapter 7.

The ASP.NET Implementation of MVC

In MVC, controllers are C# classes, usually derived from the `System.Web.Mvc.Controller` class. Each `public` method in a class derived from `Controller` is an *action method*, which is associated with a configurable URL through the ASP.NET routing system. When a request is sent to the URL associated with an action method, the statements in the controller class are executed in order to perform some operation on the domain model and then select a view to display to the client. Figure 3-1 shows the interactions between the controller, model, and view.

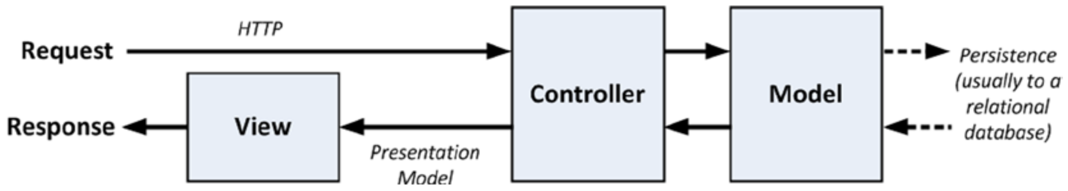


Figure 3-1. *The interactions in an MVC application*

The ASP.NET MVC Framework uses a *view engine*, which is the component responsible for processing a view in order to generate a response for the browser. Earlier versions of MVC used the standard ASP.NET view engine, which processed ASPX pages using a streamlined version of the Web Forms markup syntax. MVC 3 introduced the Razor view engine, which was refined in MVC 4 (and unchanged in MVC5) and that uses a different syntax entirely, which I describe in Chapter 5).

■ **Tip** Visual Studio provides IntelliSense support for Razor, making it a simple matter to inject and respond to view data supplied by the controller.

ASP.NET MVC doesn't apply any constraints on the implementation of your domain model. You can create a model using regular C# objects and implement persistence using any of the databases, object-relational mapping frameworks, or other data tools supported by .NET.

Comparing MVC to Other Patterns

MVC is not the only software architecture pattern, of course. There are many others and some of them are, or at least have been, extremely popular. We can learn a lot about MVC by looking at the alternatives. In the following sections, I briefly describe different approaches to structuring an application and contrast them with MVC. Some of the patterns are close variations on the MVC theme, whereas others are entirely different.

I am not suggesting that MVC is the perfect pattern for all situations. I am a proponent of picking the best approach to solve the problem at hand. As you will see, there are situations where some competing patterns are as useful as or better than MVC. I encourage you to make an *informed* and *deliberate* choice when selecting a pattern. The fact that you are reading this book suggests that you already have a certain commitment to the MVC pattern, but it is always helpful to maintain the widest possible perspective.

Understanding the Smart UI Pattern

One of the most common design patterns is known as the *smart user interface* (smart UI). Most programmers have created a smart UI application at some point in their careers—I certainly have. If you have used Windows Forms or ASP.NET Web Forms, you have too.

To build a smart UI application, developers construct a user interface, often by dragging a set of *components* or *controls* onto a design surface or canvas. The controls report interactions with the user by emitting events for button presses, keystrokes, mouse movements, and so on. The developer adds code to respond to these events in a series of *event handlers*: small blocks of code that are called when a specific event on a specific component is emitted. This creates a monolithic application, as shown in Figure 3-2. The code that handles the user interface and the business is all mixed together with no separation of concerns at all. The code that defines the acceptable values for a data input, that queries for data or modifies a user account, ends up in little pieces, coupled together by the order in which events are expected.

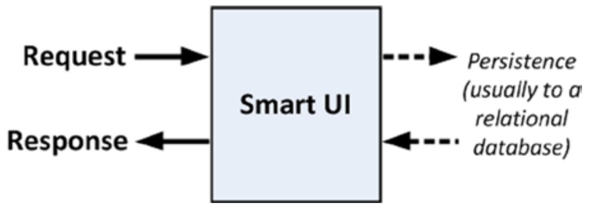


Figure 3-2. *The Smart UI pattern*

Smart UIs are ideal for simple projects because you can get some good results fast (by comparison to MVC development which, as you'll see in Chapter 7, requires some careful preparation and initial investment before getting results). Smart UIs are also suited to user interface prototyping. These design surface tools can be *really* good, although I always find the Web Forms design surface in Visual Studio to be awkward and unpredictable. If you are sitting with a customer and want to capture the requirements for the look and flow of the interface, a Smart UI tool can be a quick and responsive way to generate and test different ideas.

The biggest drawback is that Smart UIs are difficult to maintain and extend. Mixing the domain model and business logic code in with the user interface code leads to duplication, where the same fragment of business logic is copied and pasted to support a newly added component. Finding all of the duplicate parts and applying a fix can be difficult. It can be almost impossible to add a new feature without breaking an existing one. Testing a Smart UI application can also be difficult. The only way is to simulate user interactions, which is far from ideal and a difficult basis from which to provide full test coverage.

In the world of MVC, the Smart UI is often referred to as an *anti-pattern*: something that should be avoided at all costs. This antipathy arises, at least in part, because people come to MVC looking for an alternative after spending part of their careers trying to develop and maintain Smart UI applications.

Although it is a common point of view, it is overly simplistic and it is a mistake to reject the Smart UI pattern out of hand. Not everything is rotten in the Smart UI pattern and there are positive aspects to this approach. Smart UI applications are quick and easy to develop. The component and design tool producers have put a lot of effort into making the development experience a pleasant one, and even the most inexperienced programmer can produce something professional-looking and reasonably functional in just a few hours.

The biggest weakness of Smart UI applications—maintainability—doesn't arise in small development efforts. If you are producing a simple tool for a small audience, a Smart UI application can be a perfect solution. The additional complexity of an MVC application simply isn't warranted.

Understanding the Model-View Architecture

The area in which maintenance problems tend to arise in a Smart UI application is in the business logic, which ends up so diffused across the application that making changes or adding features becomes a fraught process. An improvement in this area is offered by the *model-view* architecture, which pulls out the business logic into a separate domain model. In doing this, the data, processes, and rules are all concentrated in one part of the application, as shown in Figure 3-3.

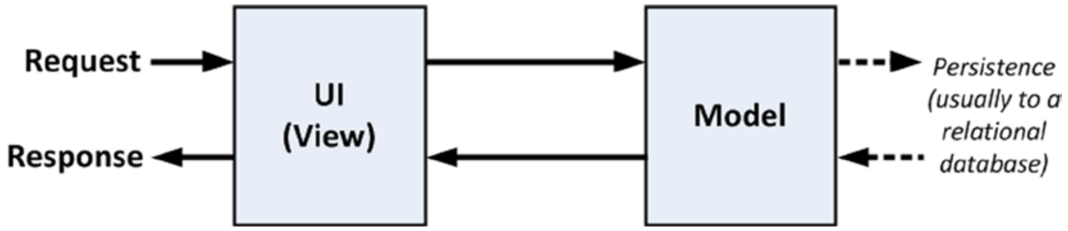


Figure 3-3. *The model-view pattern*

The model-view architecture can be an improvement over the monolithic Smart UI pattern—it is much easier to maintain, for example–but two problems arise. The first is that since the UI and the domain model are closely integrated, it can be difficult to perform unit testing on either. The second problem arises from practice, rather than the definition of the pattern. The model typically contains a mass of data access code—this need not be the case, but it usually is—and this means that the data model does not contain just the business data, operations, and rules.

Understanding Classic Three-Tier Architectures

To address the problems of the model-view architecture, the *three-tier* or *three-layer* pattern separates the persistence code from the domain model and places it in a new component called the *ddata access layer* (DAL). This is shown in Figure 3-4.

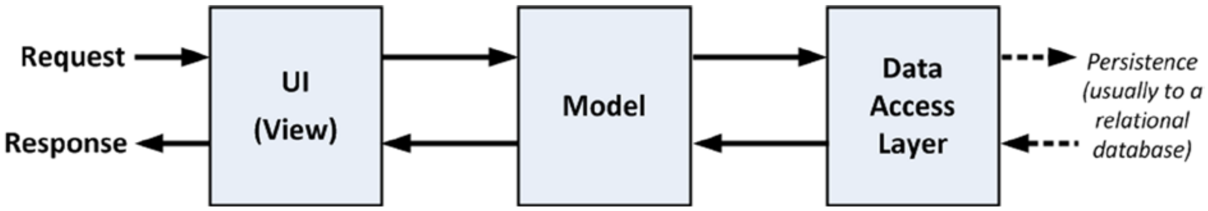


Figure 3-4. *The three-tier pattern*

The three-tier architecture is the most widely used pattern for business applications. It has no constraints on how the UI is implemented and provides good separation of concerns without being too complicated. And, with some care, the DAL can be created so that unit testing is relatively easy. You can see the obvious similarities between a classic three-tier application and the MVC pattern. The difference is that when the UI layer is directly coupled to a click-and-event GUI framework (such as Windows Forms or ASP.NET Web Forms), it becomes almost impossible to perform automated unit tests. And because the UI part of a three-tier application can be complex, there's a lot of code that can't be rigorously tested.

In the worst scenario, the three-tier pattern's lack of enforced discipline in the UI tier means that many such applications end up as thinly disguised Smart UI applications, with no real separation of concerns. This gives the worst possible outcome: an untestable, unmaintainable application that is excessively complex.

Understanding Variations on MVC

I have already described the core design principles of MVC applications, especially as they apply to the ASP.NET MVC implementation. Others interpret aspects of the pattern differently and have added to, adjusted, or otherwise adapted MVC to suit the scope and subject of their projects. In the following sections, I provide a brief overview of the two most prevalent variations on the MVC theme. Understanding these variations is not essential to working with ASP.NET MVC and I have included this information just for completeness because you will hear the terms used in most discussions of software patterns.

Understanding the Model-View-Presenter Pattern

Model-view-presenter (MVP) is a variation on MVC that is designed to fit more easily with stateful GUI platforms such as Windows Forms or ASP.NET Web Forms. This is a worthwhile attempt to get the best aspects of the Smart UI pattern without the problems it usually brings.

In this pattern, the presenter has the same responsibilities as an MVC controller, but it also takes a more direct relationship to a stateful view, directly managing the values displayed in the UI components according to the user's inputs and actions. There are two implementations of this pattern:

- The *passive view* implementation, in which the view contains no logic—it is a container for UI controls that are directly manipulated by the presenter.
- The *supervising controller* implementation, in which the view may be responsible for some elements of presentation logic, such as data binding, and has been given a reference to a data source from the domain models.

The difference between these two approaches relates to how intelligent the view is. Either way, the presenter is decoupled from the GUI framework, which makes the presenter logic simpler and suitable for unit testing.

Understanding the Model-View-View Model Pattern

The *model-view-view model* (MVVM) pattern is the most recent variation on MVC. It originated from Microsoft and is used in the Windows Presentation Foundation (WPF). In the MVVM pattern, models and views have the same roles as they do in MVC. The difference is the MVVM concept of a *view model*, which is an abstract representation of a user interface—typically a C# class that exposes both properties for the data to be displayed in the UI and operations on the data that can be invoked from the UI. Unlike an MVC controller, an MVVM view model has no notion that a view (or any specific UI technology) exists. An MVVM view uses the WPF *binding* feature to bi-directionally associate properties exposed by controls in the view (items in a drop-down menu, or the effect of pressing a button) with the properties exposed by the view model.

■ **Tip** MVC also uses the term *view model* but refers to a simple model class that is used only to pass data from a controller to a view, as opposed to *domain models*, which are sophisticated representations of data, operations, and rules.

Building Loosely Coupled Components

One of most important features of the MVC pattern is that it enables separation of concerns. I want the components in my applications to be as independent as possible and to have as few interdependencies as I can arrange.

In an ideal situation, each component knows nothing about any other component and only deals with other areas of the application through abstract interfaces. This is known as *loose coupling*, and it makes testing and modifying applications easier.

A simple example will help put things in context. If I am writing a component called `MyEmailSender` that will send e-mails, I would implement an interface that defines all of the public functions required to send an e-mail, which I would call `IEmailSender`.

Any other component of my application that needs to send an e-mail—let's say a password reset helper called `PasswordResetHelper`—can then send an e-mail by referring only to the methods in the interface. There is no direct dependency between `PasswordResetHelper` and `MyEmailSender`, as shown by Figure 3-5.

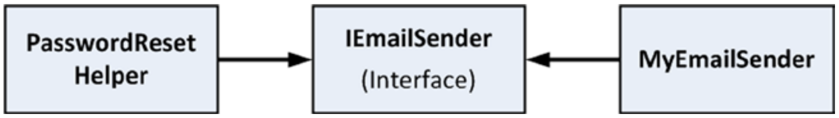


Figure 3-5. *Using interfaces to decouple components*

By introducing `IEmailSender`, I ensure that there is no direct dependency between `PasswordResetHelper` and `MyEmailSender`. I could replace `MyEmailSender` with another e-mail provider or even use a mock implementation for testing purposes without needing to make any changes to `PasswordResetHelper`. (I introduce mock implementations later in this chapter and return to them again in Chapter 6).

Using Dependency Injection

Interfaces help decouple components, but I still face a problem: C# doesn't provide a built-in way to easily create objects that implement interfaces, except to create an instance of the concrete component with the `new` keyword. I end up with code like this:

```
public class PasswordResetHelper {

    public void ResetPassword() {

        IEmailSender mySender = new MyEmailSender();

        //...call interface methods to configure e-mail details...

        mySender.SendEmail();
    }
}
```

This undermines my goal of being able to replace `MyEmailSender` without having to change `PasswordReset` helper and means that I am only part of the way to loosely coupled components. The `PasswordResetHelper` class is configuring and sending e-mails through the `IEmailSender` interface, but to create an object that implements that interface, it had to create an instance of `MyEmailSender`. In fact, I have made things worse for myself because `PasswordResetHelper` now depends on the `MyEmailSender` class *and* the `IEmailSender` interface, as shown in Figure 3-6.

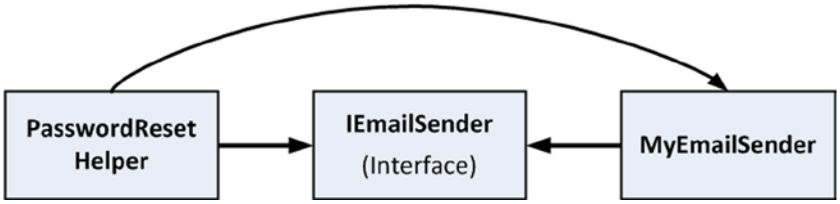


Figure 3-6. *Components which are tightly coupled after all*

What I need is a way to get objects that implement an interface *without* having to create the object directly. The solution to this problem is called *dependency injection* (DI), also known as *Inversion of Control* (IoC).

DI is a design pattern that completes the loose coupling process. As I describe DI, you might wonder what the fuss is about, but bear with me—this is an important concept that is central to effective MVC development and it can cause a lot of confusion.

Breaking and Declaring Dependencies

There are two parts to the DI pattern. The first is that I remove any dependencies on concrete classes from my component—in this case `PasswordResetHelper`. I do this by creating a class constructor that accepts implementations of the interfaces I need as arguments, like this:

```
public class PasswordResetHelper {
    private IEmailSender emailSender;

    public PasswordResetHelper(IEmailSender emailSenderParam) {
        emailSender = emailSenderParam;
    }

    public void ResetPassword() {
        // ...call interface methods to configure e-mail details...
        emailSender.SendEmail();
    }
}
```

The constructor for the `PasswordResetHelper` class is now said to *declare a dependency* on the `IEmailSender` interface, meaning that it can't be created and used unless it receives an object that implements the `IEmailSender` interface. In declaring its dependency, the `PasswordResetHelper` class no longer has any knowledge of `MyEmailSender`, it only depends on the `IEmailSender` interface. In short, the `PassworsResetHelper` no longer knows or cares how the `IEmailSender` interface is implemented.

Injecting Dependencies

The second part of the DI pattern is to *inject* the dependencies declared by the `PasswordResetHelper` class when I create instances of it, hence the term *dependency injection*.

All this really means is that I need to decide which class that implements the `IEmailSender` interface I am going to use, create an object from that class and then pass the object as an argument to the `PasswordResetHelper` constructor.

■ **Note** The `PasswordResetHelper` class declares its dependencies through its constructor. This is known as *constructor injection*. I could also declare dependencies to be injected through a public property, known as *setter injection*.

The dependencies are injected into the `PasswordResetHelper` at runtime; that is to say, an instance of some class that implements the `IEmailSender` interface will be created and passed to the `PasswordResetHelper` constructor during instantiation. There is no compile-time dependency between `PasswordResetHelper` and any class that implements the interfaces it depends on.

Because the dependencies are dealt with at runtime, I can decide which interface implementations are going to be used when I run the application. I can choose between different e-mail providers or inject a special mocked implementation for testing. Dependency injection lets me achieve the relationships I was aiming for in Figure 3-5.

Using a Dependency Injection Container

I have resolved my dependency issue, but how do I instantiate the concrete implementation of interfaces without creating dependencies somewhere else in the application? As it stands, I still have to have statements somewhere in the application like these:

```
...
IEmailSender sender = new MyEmailSender();
helper = new PasswordResetHelper(sender);
...
```

The answer is to use a *dependency injection container*, also known as an IoC container. This is a component that acts as a broker between the dependencies that a class like `PasswordResetHelper` declares and the classes that can be used to resolve those dependencies, such as `MyEmailSender`.

I register the set of interfaces or abstract types that my application uses with the DI container, and specify which implementation classes should be instantiated to satisfy dependencies. So, I would register the `IEmailSender` interface with the container and specify that an instance of `MyEmailSender` should be created whenever an implementation of `IEmailSender` is required.

When I want a `PasswordResetHelper` object in my application, I ask the DI container to create one for me. It knows that the `PasswordResetHelper` has declared a dependency on the `IEmailSender` interface and it knows that that I have specified that I want to use the `MyEmailSender` class as the implementation of that interface. The DI container puts these two pieces of information together, creates the `MyEmailSender` object and then uses it as an argument to create a `PasswordResetHelper` object, which I am then able to use in the application.

■ **Note** It is important to note that I no longer create the objects in my application myself using the `new` keyword. Instead, I go to the DI container and request the objects I need. It can take a while to get used to this when you are new to DI, but as you'll see, the MVC Framework provides some features to make the process simpler.

I do not need to write my own DI container—there are some great open source and freely licensed implementations available. The one I like and use in my own projects is called Ninject and you can get details at `www.ninject.org`. I'll introduce you to using Ninject in Chapter 6 and show you how to install the package using NuGet.

■ **Tip** Microsoft has created its own DI container, called Unity. I are going to use Ninject, however, because I like it and to demonstrate the ability to mix and match tools when using MVC. If you want more information about Unity, see `unity.codeplex.com`.

The role of a DI container may seem simple and trivial, but that is not the case. A good DI container, such as Ninject, has some clever features:

- *Dependency chain resolution*: If you request a component that has its own dependencies (e.g., constructor parameters), the container will satisfy those dependencies, too. So, if the constructor for the `MyEmailSender` class requires an implementation of the `INetworkTransport` interface, the DI container will instantiate the default implementation of that interface, pass it to the constructor of `MyEmailSender` and return the result as the default implementation of `IEmailSender`.

- *Object lifecycle management*: If you request a component more than once, should you get the same instance each time or a fresh new instance? A good DI container will let you configure the lifecycle of a component, allowing you to select from predefined options including *singleton* (the same instance each time), *transient* (a new instance each time), *instance-per-thread*, *instance-per-HTTP-request*, *instance-from-a-pool*, and many others.
- *Configuration of constructor parameter values*: If the constructor for my implementation of the `INetworkTransport` interface requires a string called `serverName`, for example, you should be able to set a value for it in your DI container configuration. It is a crude but simple configuration system that removes any need for your code to pass around connection strings, server addresses, and so forth.

Writing your own DI container is an excellent way to understand how C# and .NET handle types and reflection and I recommend it as a good project for a rainy weekend. But don't be tempted to deploy your code in a real project. Writing a reliable, robust and high-performance DI container is difficult and you should find a proven and tested package to use. I like Ninject, but there are plenty of others available and you are sure to find something that suits your development style.

Getting Started with Automated Testing

The ASP.NET MVC Framework is designed to make it as easy as possible to set up automated tests and use development methodologies such as test-driven development (TDD), which I explain later in this chapter. ASP.NET MVC provides an ideal platform for automated testing and Visual Studio has some solid testing features. Between them they make designing and running tests simple and easy.

In broad terms, Web application developers today focus on two kinds of automated testing. The first is *unit testing*, which is a way to specify and verify the behavior of individual classes (or other small units of code) in isolation from the rest of the application. The second type is *integration testing*, which is a way to specify and verify the behavior of multiple components working together, up to and including the entire Web application.

Both kinds of testing can be valuable in Web applications. Unit tests, which are simple to create and run, are brilliantly precise when you are working on algorithms, business logic, or other back-end infrastructure. The value of integration testing is that it can model how a user will interact with the UI, and can cover the entire technology stack that your application uses, including the Web server and database. Integration testing tends to be better at detecting new bugs that have arisen in old features; this is known as *regression testing*.

Understanding Unit Testing

In the .NET world, you create a separate test project in your Visual Studio solution to hold *test fixtures*. This project will be created when you first add a unit test, or can be set up automatically when you use an MVC project template. A *test fixture* is a C# class that defines a set of test methods: one method for each behavior you want to verify. A test project can contain multiple test fixture classes.

GETTING THE UNIT TEST FEVER

Being able to perform unit testing is one of the benefits of working with the MVC Framework, but it isn't for everyone and I have no intention of pretending otherwise. If you have not encountered unit testing before, then I encourage you to give it a try and see how it works out.

I like unit testing and I use it in my own projects, but not all of them and not as consistently as you might expect. I tend to focus on writing unit tests for features and functions that I know will be hard to write and that are likely to be the source of bugs in deployment. In these situations, unit testing helps me structure my thoughts about how to best implement what I need. I find that just thinking about what I test helps throw up ideas about potential problems–and that's before I start dealing with actual bugs and defects.

That said, unit testing is a tool and not a religion and only you know how much testing–and what kind of testing–you require. If you don't find unit testing useful or if you have a different methodology that suits you better, then don't feel you need to unit test just because it is fashionable. (Although if you *don't* have a better methodology and you are not testing at all, then you are probably letting users find your bugs and you are officially a *bad person.* You don't have to unit test, but you really should do *some* testing of *some* kind).

■ **Note** I show you how to create a test project and populate it with unit tests in Chapter 6. The goal for this chapter is just to introduce the concept of unit testing and give you an idea of what a test fixture looks like and how it is used.

To get started, I have created a class from an imaginary application, as shown in Listing 3-1. The class is called `AdminController` and it defines the `ChangeLoginName` method, which allows my imaginary users to change their passwords.

Listing 3-1. The Definition of the AdminController Class

```
using System.Web.Mvc;

namespace TestingDemo {

    public class AdminController : Controller {
        private IUserRepository repository;

        public AdminController(IUserRepository repo) {
            repository = repo;
        }

        public ActionResult ChangeLoginName(string oldName, string newName) {
            User user = repository.FetchByLoginName(oldName);
            user.LoginName = newName;
            repository.SubmitChanges();
            // render some view to show the result
            return View();
        }
    }
}
```

■ **Tip** I created the classes for this demonstration in a new Visual Studio project called `TestingDemo`. You don't need to recreate the examples in this section to follow along but I have included the project in the source code download available from `apress.com`.

The controller relies on some model classes and an interface, which you can see in Listing 3-2. Once again, these are not from a real project and I have simplified these classes to make demonstrating the test easier. I am not suggesting that you create user classes that have just a single string property, for example.

Listing 3-2. The Model Classes and Interface that the AdminController Relies On

```
namespace TestingDemo {

    public class User {
        public string LoginName { get; set; }
    }

    public interface IUserRepository {
        void Add(User newUser);
        User FetchByLoginName(string loginName);
        void SubmitChanges();
    }

    public class DefaultUserRepository : IUserRepository {

        public void Add(User newUser) {
            // implement me
        }

        public User FetchByLoginName(string loginName) {
            // implement me
            return new User() { LoginName = loginName };
        }

        public void SubmitChanges() {
            // implement me
        }
    }
}
```

The `User` class represents a user in my application. Users are created, managed and stored through a repository whose functionality is defined by the `IUserRepository` interface and there is a partially complete implementation of this interface in the `DefaultUserRepository` class.

My goal in this section is to write a unit test for the functionality provided by the `ChangeLoginName` method defined by the `AdminController`, as shown in Listing 3-3.

Listing 3-3. A Test Fixture for the AdminController.ChangeLoginName Method

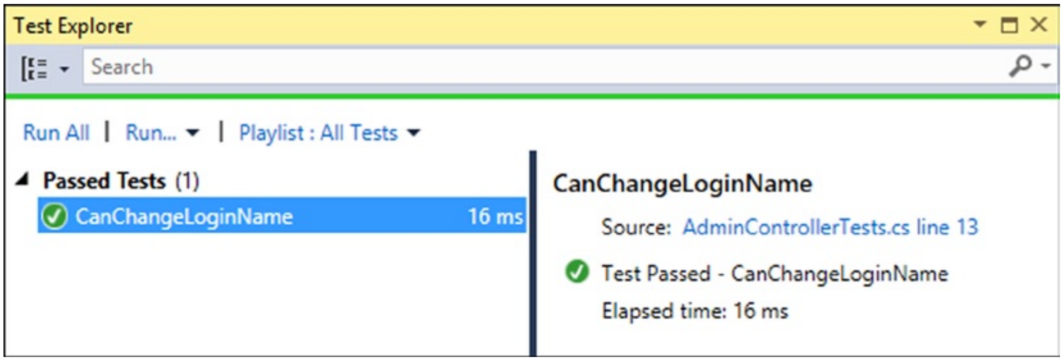
```
using System.Collections.Generic;
using System.Linq;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace TestingDemo.Tests {

    [TestClass]
    public class AdminControllerTests {

        [TestMethod]
        public void CanChangeLoginName() {
            // Arrange (set up a scenario)
            User user = new User() { LoginName = "Bob" };
            FakeRepository repositoryParam = new FakeRepository();
            repositoryParam.Add(user);
            AdminController target = new AdminController(repositoryParam);
            string oldLoginParam = user.LoginName;
            string newLoginParam = "Joe";

            // Act (attempt the operation)
            target.ChangeLoginName(oldLoginParam, newLoginParam);

            // Assert (verify the result)
            Assert.AreEqual(newLoginParam, user.LoginName);
            Assert.IsTrue(repositoryParam.DidSubmitChanges);
        }
    }

    class FakeRepository : IUserRepository {
        public List<User> Users = new List<User>();
        public bool DidSubmitChanges = false;

        public void Add(User user) {
            Users.Add(user);
        }

        public User FetchByLoginName(string loginName) {
            return Users.First(m => m.LoginName == loginName);
        }

        public void SubmitChanges() {
            DidSubmitChanges = true;
        }
    }
}
```

The test fixture is the `CanChangeLoginName` method. Notice that the method is decorated with the `TestMethod` attribute and that the class it belongs to—called `AdminControllerTests`—is decorated with the `TestClass` attribute. This is how Visual Studio finds the test fixture.

The `CanChangeLoginName` method follows a pattern known as *arrange/act/assert* (A/A/A). *Arrange* refers to setting up the conditions for the test, *act* refers to performing the test, and *assert* refers to verifying that the result was the one that was required. Being consistent about the structure of your unit test methods makes them easier to read, something you'll appreciate when your project contains hundreds of unit tests.

The test fixture uses a test-specific fake implementation of the `IUserRepository` interface to simulate a specific condition—in this case, when there is a single member, Bob, in the repository. Creating the fake repository and the `User` are done in the *arrange* section of the test.

Next, the method being tested—`AdminController.ChangeLoginName`—is called. This is the *act* section of the test. Finally, I check the results using a pair of `Assert` calls (this is the *assert* part of the test). The `Assert` method is provided by the Visual Studio test suite and lets me check for specific outcomes. I run the test from the Visual Studio `Test` menu and receive visual feedback about the tests as they are performed, as shown in Figure 3-7.



Passed Tests (1)
CanChangeLoginName 16 ms

CanChangeLoginName
Source: AdminControllerTests.cs line 13
Test Passed - CanChangeLoginName
Elapsed time: 16 ms

Figure 3-7. *Visual feedback on the progress of unit tests*

If the test runs without throwing any unhandled exceptions and all of the `Assert` statements pass without problems, the `Test Explorer` window shows a green light. If not, you get a red light and details of what went wrong.

■ **Note** You can see how my use of DI has helped with unit testing. I was able to create a fake implementation of the repository and inject it into the controller to create a specific scenario. I am a big fan of DI and this is one of the reasons.

It might seem like I have gone to a lot of effort to test a simple method, but it wouldn't require much more code to test something far more complex. If you find yourself considering skipping small tests like this one, remember that test fixtures help to uncover bugs that can sometimes be hidden in more complex tests. One improvement I could have made to my test is to eliminate test-specific fake classes like `FakeMembersRepository` by using a *mocking tool*—I show you how to do this in Chapter 6.

Using TDD and the Red-Green-Refactor Workflow

With test-driven development (TDD), you use unit tests to help design your code. This can be an odd concept if you are used to testing after you have finished coding, but there is a lot of sense in this approach. The key concept is a development workflow called red-green-refactor. It works like this:

- Determine that you need to add a new feature or method to your application.
- Write the test that will validate the behavior of the new feature when it is written.
- Run the test and get a red light.
- Write the code that implements the new feature.
- Run the test again and correct the code until you get a green light.
- Refactor the code if required. For example, reorganize the statements, rename the variables, and so on.
- Run the test to confirm that your changes have not changed the behavior of your additions.

This workflow is repeated for every feature you add. TDD inverts the traditional development process: you start by writing tests for the perfect implementation of a feature, knowing that the tests will fail. You then implement the feature, creating each aspect of its behavior to pass one or more tests.

This cycle is the essence of TDD. There is a lot to recommend it as a development style, not least because it makes a programmer think about how a change or enhancement should behave *before* the coding starts. You always have a clear end-point in view and a way to check that you are there. And if you have unit tests that cover the rest of your application, you can be sure that your additions have not changed the behavior elsewhere.

TDD seems a little odd when you first try it, but it is strangely empowering, and writing the tests first make you consider what a perfect implementation should do before you become biased by the techniques that you use to write the code.

The drawback of TDD is that it requires discipline. As deadlines get closer, the temptation is always to discard TDD and just start writing code or, as I have witnessed several times on projects, sneakily discard problematic tests to make code appear in better shape than it really is. For these reasons, TDD should be used in established and mature development teams where there is generally a high level of skill and discipline or in teams where there the team leads can enforce good practice, even in the face of time constraints.

■ **Tip** You can see a simple example of TDD in Chapter 6 when I demonstrate the testing tools built into Visual Studio.

Understanding Integration Testing

For Web applications, the most common approach to integration testing is *UI automation*, which means simulating or automating a Web browser to exercise the application's entire technology stack by reproducing the actions that a user would perform, such as pressing buttons, following links, and submitting forms. The two best-known open source browser automation options for .NET developers are

- *Selenium RC* (`http://seleniumhq.org/`), which consists of a Java "server" application that can send automation commands to Internet Explorer, Firefox, Safari, or Opera, plus clients for .NET, Python, Ruby, and multiple others so that you can write test scripts in the language of your choice. Selenium is powerful and mature; its only drawback is that you have to run its Java server.
- *WatiN* (`http://watin.org`), a .NET library that can send automation commands to Internet Explorer or Firefox. Its API isn't as powerful as Selenium, but it comfortably handles most common scenarios and is easy to set up. You need only reference a single DLL.

Integration testing is an ideal complement to unit testing. Although unit testing is well suited to validating the behavior of individual components at the server, integration testing lets you create tests that are client-focused, recreating the actions of a user. As a result, it can highlight problems that come from the interaction between components, hence the term *integration* testing. And because integration testing for a Web application is done through the browser, you can test that JavaScript behaviors work the way they are supposed to, something that is difficult with unit testing.

There are some drawbacks. Integration testing takes more time. It takes longer to create the tests and longer to perform them. And integration tests can be brittle. If you change the `id` attribute of an element that is checked in a test, for example, the test can (and usually will) fail.

As a consequence of the additional time and effort required, integration testing is often done at key project milestones, perhaps after a weekly source code check-in, or when major functional blocks are completed. Integration testing is every bit as useful as unit testing and it can highlight problems that unit testing cannot. The time required to set up and run integration testing is worthwhile, and I encourage you to add it to your development process.

I am not going to get into integration testing in this book. It is outside of my focus on the MVC Framework. Any web app can benefit from integration testing and there are no special features in the MVC Framework to support this activity. Integration testing is a separate art and what is true when performing integration testing on any Web application is also true for MVC.

Summary

In this chapter, I introduced you to the MVC architectural pattern and compared it to some other patterns you may have seen or heard of before. I discussed the significance of the domain model and introduced dependency injection, which allows us to decouple components to enforce a strict separation between the parts of an application. I demonstrated a simple unit test and you saw how decoupled components and dependency injection make unit testing simple and easy. In the next chapter, I describe the essential C# language features that are used in MVC Framework applications.