1

Requirements

Since in this book, we will create a fully functional web application, it makes sense to start by defining the requirements and what the application is. Keep in mind that we want to keep things simple and not create a hundred-page requirements document.

This book will document the development process of creating a web application using Test Driven Development (TDD) with ASP.NET MVC. We will develop a bulk email/newsletter distribution web application called EvenContact. In a nutshell, the application will allow users to manage and communicate to contacts using email messages.

Instead of delving into too much theory and in the process boring you, we will actually walk you through the development process using real tests and code. We will make mistakes on the way and correct them as we go — just as in real life. Fixing bugs and correcting mistakes is where TDD really shines, as it adds an extra safety net to ensure that the one line of code you added to fix a minor bug didn't unintentionally create five new critical bugs.

Problem

We have all received tons of email newsletters from all sorts of senders. A sender creates these newsletters and sends them out to a mailing list. The sender probably wants to track and report on the progress of their email campaign. They probably also want to manage their subscriber lists and give the subscribers the options to opt in and opt out. We will be creating a web-based email newsletter distribution application that will allow the sender to perform these tasks, and we shall call it EvenContact.

Design

There are three major and specific components in this application:

1. Message management (create, send, etc.)
2. Contact management (create, import, lists, etc.)
3. Reporting

The preceding list doesn't include generic and cross-cutting components such as membership, logging, billing, and so on.

Let's break each of these into smaller pieces that we can use as our requirements and to set the direction of the application.

Message *refers to the email message or newsletter that the user of the application will create and distribute to their recipients.*

A contact *is simply an email recipient or in its most basic form, an email address.*

Message Management

Our web application will need to allow us to manage messages. Management includes all actions related to this task, including creating, editing, displaying, deleting, and sending messages. The user interface for the application should be intuitive and easy to use, and the screen flow should fit with use cases for the system. For example, the user might want to send a message after creating it or might save it for later.

Create Message

The user needs the ability to create a message from scratch by going to a page that lets him compose the message and give each message a unique name. They need to be able to create a simple text message and a rich HTML message. The HTML message could contain images, tables, and standard HTML formatting such as bold, italic, and so on. HTML messages need to be created in a What You See Is What You Get (WYSIWYG) editor as well as provide the ability to edit the markup. The HTML message editor will provide functions to upload and use an image or link to an existing one.

Message Templates

The user can select from a list of predefined templates to jump start the message creation. Once a template is selected, the user will be directed to the message composition page with the message editor prefilled with the HTML from the template, where they can customize the message.

List Messages

The user can view a list of all their previously created messages. They should be able to perform actions on a selected message. Actions can be edit, delete, duplicate, and so on. The list should be in a grid format and support paging and sorting. The user can sort by date sent, date created, or message name. Only one message can be acted on at once, that is, no multiple selection.

Edit Message

The user can open a message and make changes to it. The most common scenario will be for the user to click the edit action in the message list to edit the message. Editing will provide a similar interface to the message creation view. Editing a text message will display the simple editor and editing an HTML message will display the WYSIWIG editor.

Save Message

The user can save a message without sending it. If a message has already been sent then a copy is created. The save action will be available from the edit and create screens. While in edit mode, saving will overwrite the existing message unless it has been sent before. In that case, a copy is created. The user will be prompted to name the copy.

Duplicate Message

The user can duplicate a message from a previously created message. This will create an exact copy of the message, and the user will be prompted to name the newly created message. The message name must be unique. Once a duplicate is created, it will have no relationship or association with the original message.

Send Message

The user will be able to send a message. Once a message is sent, it cannot be edited or deleted. The user is required to name the campaign before sending it. The user will have the option to send it right then or schedule a later delivery.

> Campaign *refers to an instance of sending a message. So when a user creates a message and sends it, they have created a campaign. For example, when a user creates and sends a message called "November Newsletter" to a recipient or group of recipients, that is a campaign.*

Delete Message

The user cannot delete a message that has already been sent. If a message has not been sent, then it can be deleted. Deleting a message completely removes it from the database. A delete action cannot be undone. The user must confirm the delete action.

Search Messages

The user can search their messages. The search should look into message content and metadata. Search results should display in a grid format that supports paging. Since this is a simple search and the results contain no weights, the results will be sorted by date in descending order (newest message on top).

Contact Management

Contacts are a centerpiece of this application, and the ability to manage these contacts is fundamental to the application. The user needs to create, modify, and delete contacts as well as group them in lists and target them in mailing campaigns.

Create Contact

The user can create a contact. All contacts are stored in the user's global "address book." An email address is the only required element for a contact and works as a logical unique identifier for a contact. This means that no two contacts can have the same email address.

> *The global address book is a list of all the contacts that the user owns. This is synonymous with* global list*, and the terms will be used interchangeably.*

There are several ways for a user to create a contact:

1. Enter one contact at a time
2. Enter multiple contacts in a multiline text field. One contact should be entered per line.
3. Import contacts from a text file. The text file will be comma delimited and contain one contact per line.

Edit/View Contact

The user can make changes to a contact. The email field cannot be changed (since it is the unique identifier for a contact). All other fields can be changed or cleared. Edit and View are identical except that when viewing, everything is read-only and cannot be changed.

List Contacts

The user can view all contacts and perform actions on the selected contacts. This will display the list of contacts in a grid format that can be paged and sorted. Actions will include edit, delete and view. The grid can be sorted by email, first name or last name. The user can select multiple contacts. The only action available when multiple contacts are selected is the delete action.

Delete Contact

The user can delete a contact. Deleting a contact that was used as a recipient of a previously sent message should not delete the actual record but mark it as archived. This is needed to preserve the accuracy of historical stats and to maintain the state of sent messages.

Search Contacts

This provides a way for users to search for a contact. The search will look through all contact metadata (email, name, description, etc.). The search results will be displayed in a grid format that is similar to the list of contacts and has the same features.

Create Contact List

The user can create a contact list to organize their contacts. A list can contain some or all of the user's contacts.

> *Think of a* contact list *as a mailing list. It allows the user to group contacts together to better target, segment, and organize their contacts. A user can create a contact list for Valued Customers, New Customers, and so on.*

Edit/View Contact List

The user can edit a contact list. They can edit its metadata (name, description, etc.) and add/remove users to/from the contact list. Deleting a contact from a contact list only removes the contact from that list but doesn't actually delete the contact from the "global" list of contacts. Viewing is identical to editing except that nothing can be changed.

Add Contact to Contact List

The user can add contacts to the list in two ways:

1. Using the same workflow to create a contact as mentioned previously. If a contact already exists, it will be added to the list. If a contact is new, it will be created then added to the list.
2. By selecting contacts from a list. These can come from a previously created list or the global list.

Delete Contact List

The user can delete a contact list. Deleting a contact list does not delete its contacts by default. The user will have the option to also delete the contacts. When deleting a contact the same rules apply.

Reports and Stats

The user will be able to view reports and stats on users, campaigns, and so on.

> *There are a ton of reports that can be created to analyze the data and give users more insight into their customers, but for the sake of simplicity we will focus on basic reports.*

Campaign Report

- ❑ Number of recipients
- ❑ Number of views (i.e., people who opened the email message). This only works for HTML messages
- ❑ Total number of clicks
- ❑ Links clicked and click count
- ❑ Number of unsubscribes

User Report

- ❑ Total number of emails sent to user
- ❑ Total click count
- ❑ Total view count
- ❑ Campaigns targeting user
  - ❑ Viewed?
  - ❑ Click count
  - ❑ Links clicked and click count

Miscellaneous Requirements

Here is a list of some miscellaneous requirements:

1. Emails sent must have a link to unsubscribe.
2. Users need an HTML/JavaScript snippet to provide a subscription form.

3. Accounts are free to create.
4. Users will be charged based on the number of emails sent.
5. There will be different payment plans, including a pay-per-use plan.

Solution

So far, the problem is that we need a website to manage email distribution and the design is a set of requirements that must be met to address the problem. In the solution section, I want to talk about the technology, tools, and methodology that we will be using to envision our solution.

Model-View-Controller

The Model-View-Controller (MVC) is a common architectural/design pattern that is used to isolate the business logic from the user interface. This results in a loose coupling between the two that allows us to easily modify the visual presentation (the view) or the underlying business layer independently without affecting each other. In MVC, the model represents data, the view represents the user interface, and the controller manages the communication between the user's actions on the view and the model.

ASP.NET MVC

The ASP.NET MVC framework — as the name implies — is the Model-View-Controller framework for ASP.NET. It is different than the ASP.NET WebForms model that we have been accustomed to for the past several years. In the WebForms world, we have *PostBacks*, *ViewStates*, *server-side event handling*, *server controls*, and so on. In MVC, we don't get any of that. Some might think, "What? I can't live without these things!" and others might think "It's about time." Microsoft has gone to great lengths to emphasize the fact that the MVC framework does not and will not replace WebForms. It is simply an alternative to it.

You might ask yourself, "Which one should I use?" The short answer is, "It depends." If you want to use Test Driven Development, then MVC is a much better-suited candidate. That doesn't mean you can't do TDD with WebForms, but it is so much easier with MVC. If you are creating a prototype or a simple data-driven website (an internal site, for example), then I would say WebForms is better suited for this. There is just no faster way to create a grid that is pageable, sortable, and editable than by using a grid view in a WebForms application with ASP.NET Dynamic Data.

MVC Strengths

- ❑ Absolute control over rendered HTML
- ❑ No ViewState
- ❑ No PostBack
- ❑ Clean separation of responsibility (model, view, controller)
- ❑ Better suited for TDD

WebForms Strengths

- Better suited for Rapid Application Development (RAD)
- A plethora of third-party controls and tools
- Better integration with the IDE (Visual Studio 2008)
- Abstraction of low-level technologies (HTML, CSS, JavaScript)

Some of the advantages of WebForms will eventually disappear as the MVC framework matures. For example, third-party support and IDE integration will eventually be comparable on both platforms. At the end of the day, it will be a matter of preference and personal style.

In the following pages, I want to explain some important concepts that are central and in some cases new to the ASP.NET MVC framework.

Model

The model is a domain-specific representation of the information. These are the classes that represent your entities; for example, `User`, `Customer`, `Account`, and so on. There are many ways to create a model; you can use a DataSet, LINQ to SQL, or the ADO.NET Entity Data Model. You can also use your preferred Object-Relational Mapping (ORM) tool to generate your model classes or you can just create your own POCO classes.

> *POCO stands for Plain Old CLR Object, and it came over from the Java world where it is called POJO (the J stands for Java; the CLR stands for Common Language Runtime). It is used to indicate that the object is a standard object and not a complex or special object to be used for a specific framework or component. For example, the DataTable, WebForm, and UserControl are non-POCO, while the following class is a POCO:*

```
public class User
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime DateOfBirth { get; set; }
}
```

View

If you use ASP.NET MVC out of the box, then your views will either be a `ViewPage` or a `ViewUserControl`. These are the counterparts to a `Page` and `UserControl` in the WebForm world. Usually a `ViewUserControl` is used for partial views. Just as in WebForms a `ViewPage` can inherit from a master page, in MVC you inherit from `ViewMasterPage`.

Controller

The controller is the liaison between the view and the model. The controller is made up of actions that are invoked by the view. An action performs a specific task, which most likely involves dealing with the model, and then renders the next view to show its results. Think of a scenario when a user clicks the submit button on a form (the view). The framework will route the button click (or more specifically the HTML form submit) to a specific action in a controller that receives the form data and acts on it and then returns back the results by rendering a new view. Figure 1-1 shows the relationship between the model,

view, and controller and is pretty much the standard diagram you will see whenever the model-view-controller pattern is discussed.

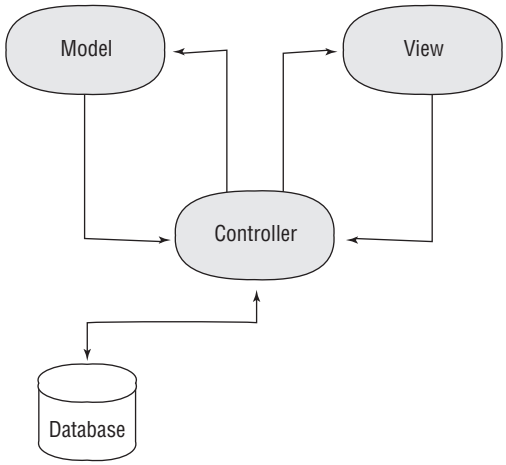

Figure 1-1

Routing

The ASP.NET MVC framework includes a flexible URL routing system that allows us to define mapping rules. The routing system maps an incoming URL to the appropriate controller and action. For example, if we use the default route definition, a URL like `/user/edit/2` will be mapped by the routing system to the Edit action in the User controller and will be passed 2 as an ID parameter. The mapping system also constructs outgoing URLs, which we use in our views to submit forms and create links and so on.

The routing system is pretty flexible and gives us the ability to change routes without changing code. So if we decide to change the URL from `/user` to `/customer`, then we just change it in our route definition in the application startup logic.

ViewData

You are probably wondering how the controller sends data to the view. For example, if you are editing a customer record, the controller needs to grab that data from our data store and send it to the view. This is done using the `ViewData` property in the controller. `ViewData` is of type `ViewDataDictionary`, and you can set/get values to/from it by either using the dictionary (e.g., `ViewData["title"] = "Edit Customer"`) or setting the `ViewData`'s Model property to an instance of your model class that you want sent to the view — or you can do both. Here is an example:

```
var model = Repository.GetCustomer(2);
ViewData.Model = model;
ViewData["title"] = "Edit Customer";
```

The view will be able to retrieve the data sent to it by accessing its own `ViewData` property. You can get the page title set above like this:

```
<title><%= Html.Encode(ViewData["Title"]) %></title>
```

And you can access the model mentioned above like this:

```
<%= Html.TextBox("firstname", Html.Encode(ViewData.Model.FirstName))%>
```

This will generate the following HTML:

```
<input id="firstname" type="text" value='Firstname' />
```

HTML Helpers

You might have noticed that, in some of the previous examples, I used the `Html` helper class. This class contains several helper methods that allow you to generate HTML easily. Some people prefer to use them, while others don't. I am somewhere in between. The following two statements are identical in that they produce identical HTML:

```
<%= Html.TextBox("email")%>
```

and

```
<input id="email" type="text" />
```

Which one to use is really your choice, but in this book, you might see me mix them together. I do that because sometimes the helper method might be less efficient (or uglier) than just plain HTML, and in other instances it might be a significant time saver. The preceding example is too simple to illustrate the benefit of using the helpers.

Action Filters

Action filters are attributes that are used to decorate an action, a controller class, or even the entire assembly, that allow us to intercept the action(s) and perform some logic prior to or after execution. The framework comes with some predefined action filters out of the box. One such action filter is `Authorize`, which will prevent an unauthorized user from executing the action by directing them to the login page. Action filters can be used for tracing, exception handling, and logging, among other uses. To create an action filter, you inherit from `ActionFilterAttribute` and override the methods you need. Your class would look something like this:

```
public class TraceAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext filterContext)
    {
        //write trace stuff before executing the action
    }
    public override void OnActionExecuted(ActionExecutedContext filterContext)
    {
        //write trace stuff after executing the action
    }
}
```

TempData

`TempData` is similar to `ViewData` in that it allows us to send data to the view. The fundamental difference between `TempData` and `ViewData` is that `TempData` is only available for the life of one request. That makes it ideal for storing temporary information that you only want to send to the view once. Good candidates to store in `TempData` are messages that are only relevant for that specific request. For

example, if a record update was successful, you could say `TempData["msg"] = "Record updated successfully";`. The view can then retrieve this message and notify the user that the record was updated successfully. Once that is done, the message is not needed anymore, and there is no reason to persist it anywhere.

ModelState

The `ViewData` contains a property called `ModelState`. The `ModelState` is a collection of model states. Specifically it is of type `ModelStateDictionary` and contains a dictionary of `ModelState` objects accessible with a string key. This dictionary is used to pass the model state back to the view, which is very useful in validation scenarios. For example, one can add an item to the `ModelState` indicating that the login credentials were not correct and, by using helper methods, validation is greatly simplified. We will discuss this in more detail later, but here is a quick example of how you would add something to the `ModelState`:

```
ViewData.ModelState.AddModelError("password",
                password, "Password is incorrect.");
```

Methodology, Concepts, and Approach

In the next few pages, you will learn about the approach, methodology, and common patterns that we will use to create the application. These are pretty large topics and going into depth about some of them is beyond the scope of this book. Nonetheless, I want to briefly explain them, so that we have a common language to use. At the end, it should help you understand the book better.

Test Driven Development (TDD)

Test Driven Development, or TDD, is a software development technique that is made up of short iterations where tests are written first and then code is written to satisfy the test condition. To use a more concrete example, if we wanted to write a method to validate an email address, we would write the test first that covers this use case and then write the method to make the test pass.

The TDD cycle consists of the following sequence:

1. **Add a test.** Create a test to satisfy a requirement. In our example, we would create a test that calls the email validation method with an invalid email address.
2. **Run the tests.** Since we haven't written any code, the test should fail. This is an important step because it validates that the test won't always pass, which would make it worthless. Initially, you can't even run the tests because you will have compiler errors, since you haven't written the code your test is calling yet.
3. **Write code.** Write code to satisfy the test condition. In this case, you will create the `IsValidEmail` method to validate an email address.
4. **Run the tests.** The tests should now pass. If they fail, then repeat step 3 until they pass.
5. **Refactor.** Now that the requirement is met, we can refactor our code. Re-run the tests as you refactor the code to make sure that you have not broken anything.

The preceding cycle is repeated throughout the development process for each new feature that needs to be added.

If you examine the preceding cycle and think about it, you might realize its benefits. One immediate and apparent benefit is that we already have a unit test written. We have all worked on projects where we agreed to create unit tests one week before release and what ends up happening most of the time is that we are so busy trying to meet the deadline that we don't have time to create the tests.

Having a test in place is tremendously powerful and comforting. It allows us to change the code in the future, press the Run Tests button, and immediately find out if the two lines of code we just added introduced 20 new bugs.

Starting the iteration with a test means that we need to very clearly understand the requirements in order to satisfy them. Having a clear understanding of the requirements, use cases, and user stories will inevitably lead to better software and ultimately a satisfied customer.

We all know that the requirements will change a few weeks (or even days) into the project. Therefore, satisfying the requirement with tests allows us to easily change the code and verify the changes by running the tests (see Figure 1-2).

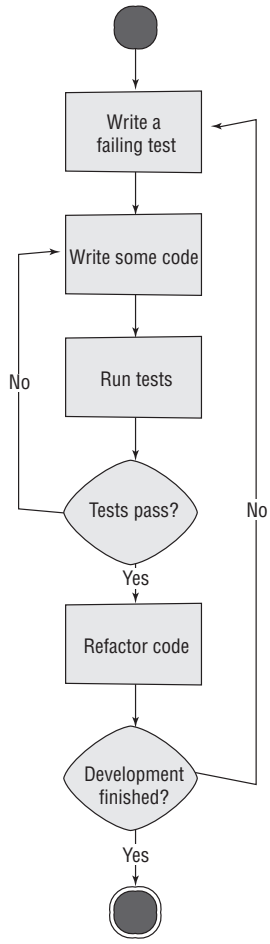

Figure 1-2

Aspect-Oriented Programming (AOP)

Aspect-oriented programming, or AOP, is a style of programming that increases code modularity, readability and reusability by allowing for the separation of cross-cutting concerns. A cross-cutting concern is a behavior that is common across your application layers and can be hard to express using traditional object-oriented techniques.

For example, a common aspect is logging. In order to log your application's actions, you have to sprinkle logging code throughout your code. However, most classes or methods that you want to log from don't (and shouldn't) really care about logging. For example, one can write a method like this:

```
public void AddFriend(string friendName)
{
    try
    {
        LogMessage("AddFriend Called");
        //code that adds a friend
    }
    catch (Exception ex)
    {
        LogException(ex);
        //handle exception
    }
    finally
    {
        LogMessage("AddFriend Ended");
    }
}
```

There are several problems with the preceding method:

- The logging code is dispersed throughout.
- The code is hard to maintain or change.
- The code has to be manually written for every method.

A better way to do this is to use aspects. Using PostSharp as our AOP framework, we can create a new aspect for logging that can be applied to our method above by simply applying an attribute. Our code might look like this:

```
[Log()]
public void AddFriend(string friendName)
{
    try
    {
        //code that adds a friend
    }
    catch (Exception ex)
    {
        //handle exception
    }
}
```

The `Log` aspect (attribute) will be able to log the appropriate messages at different points during the method invocation, for example on entry, on exit, on error, and so on.

Definitions

Aspect — A modularization of a cross-cutting concern — for example, logging, transaction management, tracing, exception handling, or authorization

Joinpoint — A point where the main program and the aspect meet during execution — for example, a method invocation or an exception being thrown

Advice — The action taken by the AOP framework at a joinpoint

Patterns

I want to briefly explain some design patterns that we will most likely use in this application. This is not a patterns book and there are people more qualified to delve into the details of each pattern; I simply want to briefly name and describe each pattern. Most of my pattern descriptions came from the canonical reference for software patterns — Martin Fowler's book *Patterns of Enterprise Application Architecture* (PoEAA; Addison-Wesley, 2002).

We have already explained the Model-View-Controller (MVC) pattern, so we will skip it in this section.

Strategy

This pattern is intended to define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Definitions

Strategy — An interface declaration that is common to the algorithm

Concrete Strategy — An implementation of the strategy

Context — An object that is configured to use a concrete strategy

Conceptually, the strategy pattern is shown in Figure 1-3.

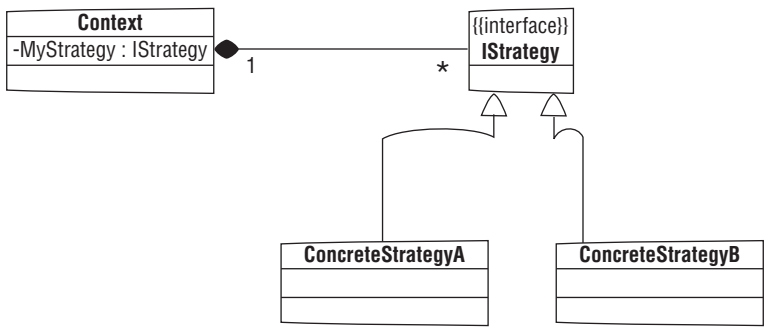

Figure 1-3

One example where the strategy pattern can be applied is in sorting. The class diagram could look something like Figure 1-4.

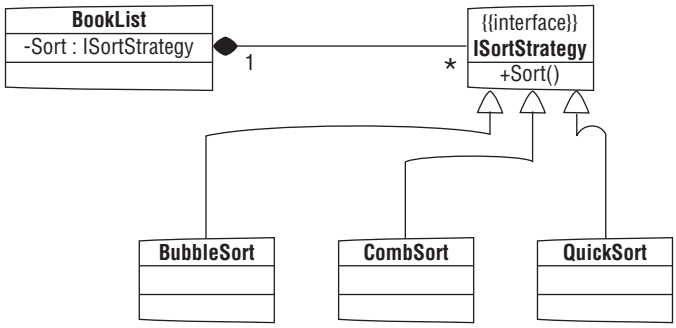

Figure 1-4

Null Object

A null object is an object designed to represent the absence of an object. If you invoke anything on a null object in .NET, you will get a `NullReferenceException`. It is better to return an object instance that represents the lack of an object. For example, if your customer doesn't have any orders and you try to get their orders, using the null object pattern, you would do this:

```
public List<Orders> GetOrders(int customerId)
{
    //get orders from database
    //if no orders exist then return an empty list
    return new List<Orders>();
}
```

This is better than returning null because the consumers of your method won't have to perform null checks, and calling the `Count` property on the returned object won't throw a `NullReferenceException`.

```
var orders = GetOrders(2);
//won't throw an exception
txtTotalOrders.Text = orders.Count;
```

Lazy Instantiation (aka Lazy Load)

This is the pattern where the creation of an object, calculation of value, or execution of an expensive process is delayed until it is first needed. For example, here is a property that will return a user's contacts, but it is only computed when it is called the first time. So if it is never called, we never incur the cost of retrieving the list of contacts.

```
private List<Contact> contacts;
public List<Contact> Contacts
{
    get
    {
        if(contacts == null)
```

```
            {
                contacts = GetContactsFromDatabase(userId);
            }
            return contacts;
        }
    }
```

Repository

This pattern allows us to create a data access layer that is independent of the data store. This makes data access changes and TDD much easier. For example, we can create a contact repository by defining the following interface:

```
public interface IContactRepository
{
    IQueryable<Contact> GetContact(long id);
    IQueryable<Contact> GetContacts(long customerId);
    long Create(string email, string name, string description);
    bool Delete(long id);
    bool Update(Contact contact);
}
```

We can now implement this interface to talk to a SQL database in production, implement it to talk to an XML file during integration testing, or even implement it to talk to an in-memory database for unit-testing purposes.

Principles

The following is a list of software principles that I will apply throughout the book whenever and wherever possible. These principles are general and abstract and have wide applicability. They create a set of programming guidelines that can be used during software development regardless of the language. Actually, many of these principles are so generic that they can be applied to other fields unrelated to software development.

Open-Closed Principle

The open-closed principle states that a software entity (class, module, assembly, etc.) should be open for extension but closed for modification. This allows others to change the entity's behavior without changing its source code. This is usually achieved through inheritance. For example, if a class has a `Send` method that sends out a message, we can open it to extension by allowing derived classes to override the `Send` method and implement a different send functionality such as sending by instant messaging (IM) instead of Simple Mail Transfer Protocol (SMTP).

YAGNI

YAGNI is short for "you ain't gonna need it" and simply means that you should only add code/functionality when you need it and not when you foresee that you need it. Recently, I have become a big fan and advocate of the YAGNI approach. That is mainly because I remember all the code that I have written in the past that I *thought* I was going to need. We all know that every line of code adds complexity, and every new feature needs to be tested, debugged, fixed, documented, and supported.

So think very hard before you add new features. Do you really need it? Do you need it now? If you don't need it now then don't do it now. When the need arises (and it might never arise), then you go ahead and add it — by then it will probably be different than what you initially thought.

KISS

The KISS (keep it simple, stupid) principle states that things should be kept simple and complexity avoided. This principle can be applied to other disciplines and not just software development. But in the context of software development we should strive to keep things simple and avoid unnecessary complexity, so we don't end up with a Rube Goldberg machine. For example, if you know for a fact that you will only be using SQL server as your data store, then there is no need to add support for other database systems or to write database-agnostic code because you ain't gonna need it.

> *Everything should be made as simple as possible, but no simpler.*
>
> — Albert Einstein

DRY

DRY is acronym for "don't repeat yourself." The DRY principle is aimed at reducing duplication. Duplication decreases maintainability because it increases the difficulty of change. It might also lead to inconsistencies and ambiguity. During refactoring is a good time to look at code that you have been copying and pasting in multiple places and refactor it into a single and authoritative location. DRY is not only useful for application code but for test code as well.

Inversion of Control (IoC) and Dependency Injection (DI)

Inversion of Control (IoC) and Dependency Injection (DI) are two principles that allow us to create loosely coupled systems with fewer dependencies. Inversion of Control is the indirection of object instantiation so that objects do not directly create other objects. Instead, an IoC `Container` will inject the dependencies into an object through construct parameters or public properties.

Imagine a class that requires the use of an `EmailService` that encapsulates email functionalities. If we use constructor injection, we can define our class like this:

```
internal class MyClass
{
    public MyClass(IEmailService emailService)
    {
        this.EmailService = emailService;
    }
}
```

Then we would configure our IoC container to tell it how to instantiate an `IEmailService` and to inject the instantiated object into the above class. If this doesn't make sense or seems too vague and theoretical, don't worry; in the coming chapters, you will see better examples.

Single Responsibility

The single responsibility principle states that a class should have one responsibility. A responsibility is considered to be a reason to change. If there are two reasons to change for a class then it should be split into two classes. Take a look at the following class, which represents a `Customer`:

```
internal class Customer
{
    public long Id { get; set; }
    public string Name { get; set; }
```

```
        public Address ShippingAddress { get; set; }

        public void Save()
        {
            //code to save the customer
        }

        public void Delete()
        {
            //code to delete the customer
        }
    }
```

At first glance, this class looks OK. The problem with the class is that it has two reasons to change. One reason is if the schema changes and you need to add new properties or change existing properties. The other reason is if the persistence changes. This could happen if you change your data store or if you normalize/denormalize your database and then you want to store `Customer` across multiple tables or one table. This class should be broken into two classes, one representing the model and another dealing with persistence.

Liskov Substitution Principle

This principle basically says that any derived class can be used to substitute for its base class without altering the correctness of the application. Look at the following classes; the Liskov substitution principle states that we can use `Airplane`, `Motorcycle`, or `Boat` wherever `Vehicle` can be used:

```
internal class Vehichle
{
}
internal class Airplane: Vehichle
{
}
internal class Motorcycle:Vehichle
{
}
internal class Boat: Vehichle
{
}
```

Convention Over Configuration

Convention Over Configuration is a design paradigm that emphasizes convention over configuration by reducing decisions (configurations) that the developer needs to make. You still have the ability to configure things and still maintain a level of flexibility, but by using the convention you gain simplicity and avoid any extra work. The ASP.NET MVC framework does a great job of applying this design paradigm throughout. Here are some examples of conventions that are used:

- All controllers are named `[`*something*`]Controller`.
- All views are in the `Views` folder.
- All public methods in a controller are considered actions.
- Form fields are mapped to action parameters with the same name.

These are just a few of the conventions; we will explore more of these conventions throughout the book, and they will be highlighted in sidebars.

Tools and Frameworks

So far we have talked about the problem, design, and solution. It is now time for us to choose the tools and frameworks that will help us implement the solution. I wanted my tool selection to be encompassing and open minded. I spent a lot of time researching the alternatives and trying to pick the one that fits my application needs. So don't think of this as an endorsement of one framework over another. Also, many of these tools are open source and are very rapidly changing so that one feature that is missing at the time of my research might be implemented by the time you read this book. I think the concepts are more important than the actual tools. One thing I have to admit is that it was extremely exciting and frustrating at the same time to be able to work with cutting-edge frameworks that aren't even in beta. The good news is that most of these tools are being actively developed and have great community support. Worst case scenario, I have access to the code and, if I don't like something, I can change it.

Unit-Testing Framework

If we are going to practice TDD, we are going to need a unit testing framework. Microsoft Visual Studio Team System 2008 Edition contains a unit testing framework usually referred to as MSTest. I decided not to use MSTest even though it was in my comfort zone. My decision was solely based on the fact that I didn't want to assume that you have a few thousand dollars to spare on a copy of Visual Studio. I looked into nUnit, MbUnit, and MSTest and decided to go with MbUnit. One of the reasons was the overwhelming and enthusiastic support of several influential developers. Personally, one feature that I really liked was Row Tests, which allows us to have a single test with multiple sets of data. For example, the test below will run three times to test three different invalid usernames:

```
[Test]
[Row("abc", Description = "Username is too short")]
[Row("1user", Description = "Username starts with non-alpha")]
[Row("user$abc", "Username contains invalid characters")]
public void Register_Should_Fail_For_Invalid_Usernames(string invalidUsername)
{
    //test registration process
}
```

MbUnit also works very well with ReSharper's test runner, which is well integrated into Visual Studio, and I can easily click on the marker next to every test to run or debug it, as shown in Figure 1-5.

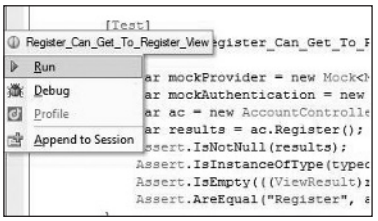

Figure 1-5

The test runner has a very good user interface that quickly pinpoints the status of the test(s) and points out failed tests as well as the reason for failure, as shown in Figure 1-6.

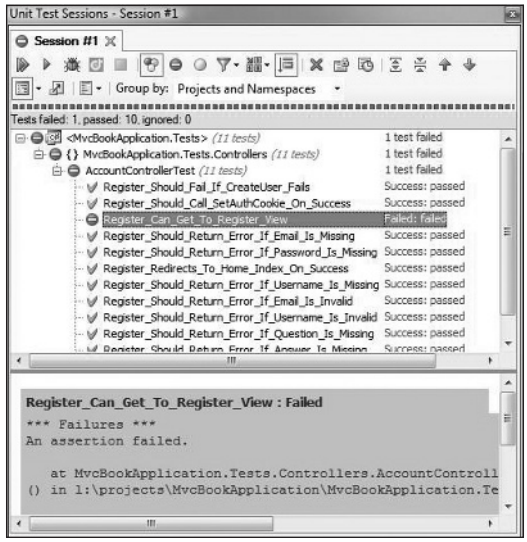

Figure 1-6

Code Coverage

Code coverage is an essential metric in Test Driven Development. It measures the degree to which the source code is tested, that is, covered. Using a code coverage tool, we can run our tests and look at our code coverage report. Ideally, you want to reach 100% code coverage. Most tools can help visually show you what is covered and what is not by highlighting code blocks in different colors. Figure 1-7 shows the code coverage window that is part of Visual Studio 2008.

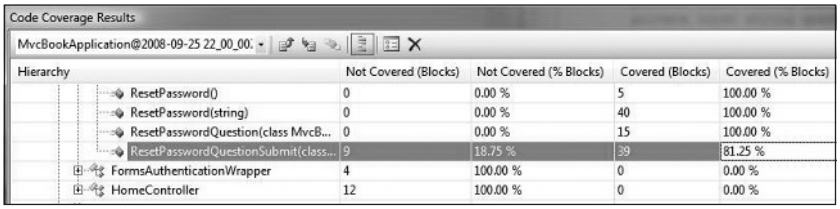

Figure 1-7

Code coverage is only available in the Developer, Test, and Team Suite editions of Visual Studio Team System 2008.

You can see from Figure 1-7 that `ResetPasswordQuestionSubmit` method has nine blocks, or 18.75%, not covered. Double-clicking the item will display the method with the code highlighted in different colors to indicate blocks that have not been covered by the test. As shown in Figure 1-8, the first two `if` statements are never reached, which means my tests are not testing these conditions.

```
[AcceptVerbs("post")]
[ActionName("ResetPasswordQuestion")]
public ActionResult ResetPasswordQuestionSubmit([ModelB
{
    ViewData["Title"] = "Reset Password";
    if (string.IsNullOrEmpty(model.Username))
    {
        ViewData.ModelState.AddModelError("username", "
                                          "Username is
    }

    if (string.IsNullOrEmpty(model.Question))
    {
        ViewData.ModelState.AddModelError("question", "
                                          "Question is
    }

    if (string.IsNullOrEmpty(model.Answer))
    {
```

Figure 1-8

If you don't have code coverage in your Visual Studio edition, then you can use NCover, which is an excellent alternative.

JavaScript Library

We will need to use JavaScript and AJAX to enhance the usability and performance of the website. It is imperative that we use a JavaScript library that is fast, easy to learn, and easy to write. After reviewing several JavaScript libraries, I immediately fell in love with and have become a big fan of JQuery. JQuery allows you to do things in JavaScript that you either never thought were possible or are very hard to do. It has a very powerful and clean syntax and allows you to do things like this:

```
$(document).ready(function() {
   $("h2").addClass("red");
 });
```

The preceding code will add the class "red" to any "h2" element. JQuery also has a huge community, good documentation, and a great repository of plug-ins and extensions.

With that said, I also really like YUI (Yahoo! User Interface Library); it has an excellent set of user interface controls/widgets. It is very well documented, has good support, and looks great. There are two components that I will use from YUI: the Rich Text Editor and the YUI Test Utility. The YUI Test Utility comes with a good test runner (shown in Figure 1-9) and has excellent documentation.

Figure 1-9

IoC Container

I wanted to use an IoC container to simplify my controllers and reduce their dependencies on other classes. This allows for vastly simplified test writing and a better design overall. There are several IoC containers out there, and they all looked very good and very powerful. But the majority had a very steep learning curve and a convoluted configuration process. I looked into the following:

- StructureMap
- Spring.Net
- Castle Windsor
- Autofac
- Ninject

Initially, I went with Autofac for its very fluent interface, highly discoverable API and very readable code. Here is a short Autofac example:

```
var builder = new ContainerBuilder();
builder.Register<Straight6TwinTurbo>().As<IEngine>();
```

After playing around with Autofac for a while, someone recommended that I check Ninject. It looked as good and as easy as Autofac, but it had better documentation and seemed slightly easier to learn. Ninject also had a very easy contextual binding syntax that lets you do this:

```
Bind<IService>().To<RedImpl>()
    .Only(When.Context
              .Target.HasAttribute<RedAttribute>());
Bind<IService>()
    .To<BlueImpl>()
    .Only(When.Context
              .Target.HasAttribute<BlueAttribute>());
```

In the following code `ConsumerA` will be injected with the `RedImpl` implementation of the `IService` interface, while `ConsumerB` will be injected with `BlueImpl` implementation.

```
class ConsumerA {
  public ConsumerA([Red] IService service) {
  }
}

class ConsumerB {
  public ConsumerB([Blue] IService service) {
  }
}
```

Choosing an IoC container was challenging and relatively subjective, but at the end of the day, I think if you choose any of these IoC containers, you will be fine.

> *One thing I love about ASP.NET MVC is the choices. You are not locked into using one vendor/product/framework and can pretty much plug in your vendor/product/framework of choice at every level of the framework. You can replace your controller factory, view engine, view rendering, route handler, and other extension points. It is very extensible.*

Mocking

Mocking allows us to simulate the behavior of a complex object in unit tests. This is extremely valuable when using a real object is difficult or impossible to do. If you wanted to unit test the create user action, but don't want the code to create a record in the database, then you can mock the calls to the database. In the context of this test, we don't really care about the implementation of the database call; instead, we want to test the create user action's logic.

There are several very good mocking libraries, including:

- TypeMock
- NMock
- NMock2
- Rhino Mocks

In my opinion, none of them is as easy to use and as elegant as Moq. It is so easy to use that I was up and running in no time. Here is how easy it is to set up a mock:

```
var mockMembership = new Mock<MembershipProvider>();
mockMembership.Expect(p => p.MinRequiredPasswordLength).Returns(4);
```

In the preceding example, we created a mocked instance of `MembershipProvider` and told it that if `MinRequiredPasswordLength` is called, it should return 4. Now you can easily pass this mocked object to your controller constructor being tested without caring about the actual implementation of the `MembershipProvider`.

Moq as explained on its homepage "is the only mocking library for .NET developed from scratch to take full advantage of .NET 3.5 (i.e., Linq expression trees) and C# 3.0 features (i.e., lambda expressions) that make it the most productive, type-safe and refactoring-friendly mocking library available. And it supports mocking interfaces as well as classes. Its API is extremely simple and straightforward, and doesn't require any prior knowledge or experience with mocking concepts."

Why EvenContact?

I wanted to create a somewhat generic application that will benefit the readers with their real-world applications and help illustrate the power and flexibility of the ASP.NET MVC framework and the test-driven approach to development. I thought this would make a good application from a technical point of view. Plus, I also got tired of reading books, articles, and blog posts that use blog engines or some sort of content management system as the example to teach everything. The blog engine has become the "hello world" of today's programming book. Last, in order to kill two birds with one stone, I wanted to create a service that I can actually offer online and potentially generate revenue from. This has a two-fold advantage. I have to make sure I am creating something good that people will want to pay for, and by doing so, I am able to produce a better book and hopefully help others learn in the process.

Summary

Now that we have everything in place, we are ready to start coding. I want you to think of this book as a documentary. I am basically documenting my development process as I progress through it. I believe this will produce a more "real-life" experience of performing TDD with ASP.NET MVC. This means that I might make decisions in Chapter 2 that are changed in Chapter 4 because of new information or false assumptions. This should be an interesting experience for all of us and hopefully one that will highlight the benefits of TDD and good design.