Extracted from:

Test-Drive ASP.NET MVC

The Pragmatic Programmers

Covers ASP.NET MVC 2.0

Test-Drive ASP.NET MVC

Jonathan McCracken

Edited by Susannah Davidson Pfalzer

Simplicity is the ultimate sophistication.
► Leonardo da Vinci

Chapter 3

Getting Organized with MVC

Now that we understand TDD and the basics of MVC, we can start implementing the sample time management application we'll create throughout Parts II and III of the book: GetOrganized. This application will improve the speed and priority of how we get things done—it will help us get organized.

The first few chapters of Part II focus on how to use and test MVC controllers; the following chapters work through how to make the site look better using views and Ajax.

This chapter starts with an overview of what we'll be doing with GetOrganized in the upcoming chapters, and then we'll dive into test-driving MVC's create, read, update, and delete (CRUD) operations to create a simple to-do list.

3.1 Time Management with GetOrganized

GetOrganized is a web-based time management system inspired by ThinkingRock, an open source Java Swing application developed by Jeremy Moore. It helps you organize your thoughts and set up action items.[1] Both GetOrganized and ThinkingRock draw their inspiration from time management guru David Allen's book *Getting Things Done* [All02].

ThinkingRock's main screen illustrates the three steps of a *Getting Things Done* system (Figure 3.1, on the next page).

1. http://www.trgtd.com.au/

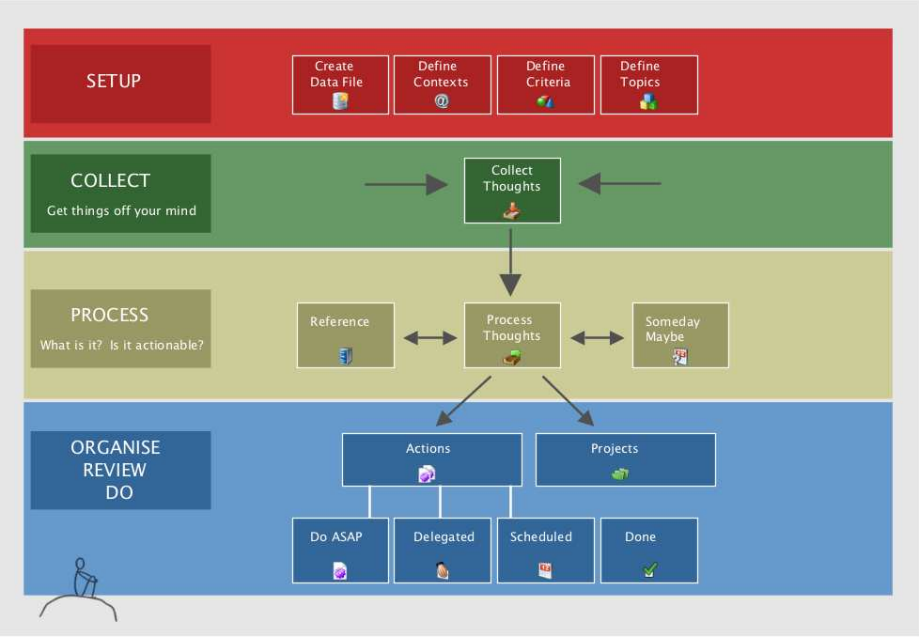

Figure 3.1: ThinkingRock helps you manage time with a three-step process: collect thoughts, process them, and implement actions.

1. Write down all the thoughts that are on your mind.
2. Process those thoughts, by either throwing them away or turning them into an action item.
3. Prioritize and complete the action items.

For the system to yield results, you commit a time every day and input your thoughts. These can be random and should have no concept of size, such as “Complete proposal for prospective client” or “Learn jQuery.” Next, you categorize these thoughts into actionable or non-actionable items. Finally, you work through those action items in the form of a to-do list.

Let’s get started by adding support for building a simple to-do list so that we can see all the things we need to work on.

3.2 Reading Data

Often the hardest thing to do when starting with TDD is to write the first test. This is especially true when a language or framework is new to us; the old pattern of writing the functional code first slips back, and before we know it, we're writing code with zero test coverage. TDD takes discipline, but we end up learning more and building greater confidence as we master it.

In the case of ASP.NET MVC, a good place to start is to test-drive the controller, because it's where so much of the application logic lives. Alternatively, you can start by test-driving your model, which we will do in Section 4.1, *Implementing Equals for Topic*, on page 75. In the end, you'll need to test both models and controllers independently.

Before we can start testing our controller, we need to create the MVC project GetOrganized.

MVC Project Structure

We installed MVC in Section 1.2, *Installing MVC*, on page 23, and this step is required before we can create a new MVC project. Once installed, we'll be able to create the solution GetOrganized with the MVC project name *Web*.

Although the project name *Web* is generic, you'll want to keep the project names simple to save screen real estate in the Visual Studio Solution Explorer. However, you'll want to modify the project properties to add a custom namespace by right-clicking the project properties. Then change the default namespace to *GetOrganized.Web*.

This is the first time we're looking at the project structure of an MVC project, so let's take a quick tour (Figure 3.2, on the following page).

By default Visual Studio generates an AccountController and HomeController. You can remove and replace these with your own code, but they give us a starting point for most web applications. The AccountController deals with user login, and the HomeController serves up the default MVC starter page. We'll touch more on the AccountController in Section 5.2, *Logging In*, on page 102.

Here's the rest of the structure:

- Content holds all images, CSS, and other static files.
- Controllers holds all your controller classes.

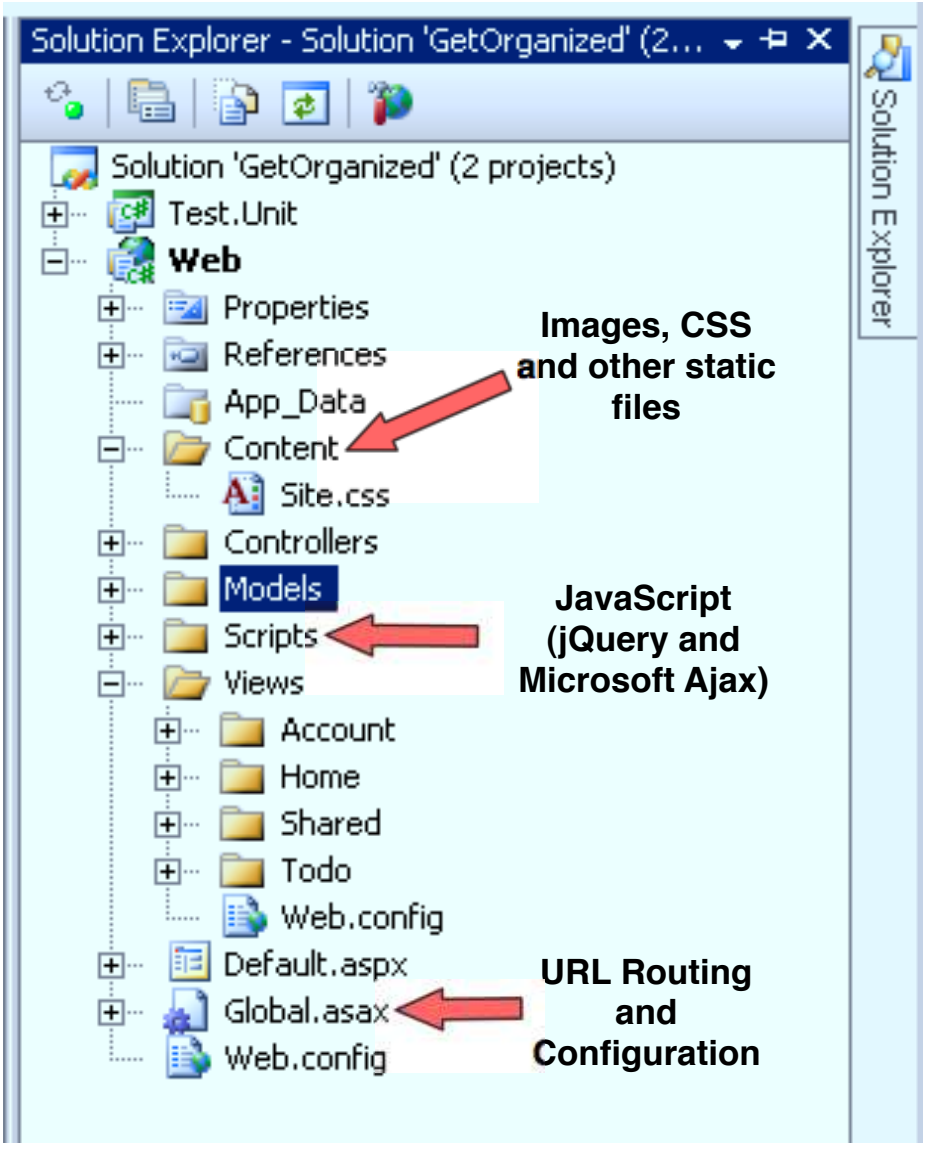

Figure 3.2: The MVC project structure has a well-defined location for all files.

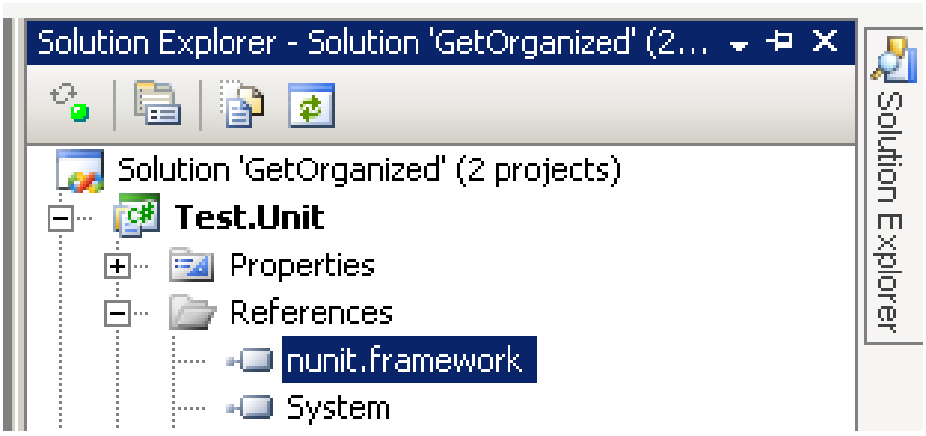

Figure 3.3: Adding a reference to NUnit is required in order to unit test.

- Models holds all your model classes.
- Views holds a subdirectory for each controller you create as well as a Shared folder for common components.
- Scripts has all a copy of jQuery and Microsoft Ajax support or any other JavaScript you create.
- Global.asax includes the routing and startup information for your application.

Our First Test

Let's get on with the business of writing our first controller test.

When creating tests, we generally create a new project, which produces a separate .NET assembly. We place our tests in that project so that test code never goes into production. We'll follow this convention by creating a project of the type Class Library, and we'll name it Test.Unit. Once it's created, make sure to add a reference to nunit.framework.dll, as shown in Figure 3.3.

This controller "should display a list of some to-do items." Hey, that sounds like a pretty good name for a test!

ReSharper Tip: Class Navigation

By naming all our test classes with the name *Test* at the end, rather than the start, it makes it easier to find the associated functional code with the test code. ReSharper has the code navigation shortcut Ctrl+N that helps us find classes in the solution. When we type in TodoController, it will bring up the actual controller as well as the test, TodoControllerTest. ReSharper's code navigation allows an even shorter form by just typing in *TC* to bring us the same result.

To start, we need to add a test class, TodoControllerTest:

Download gettingorganized/TodoControllerTest.cs

```
using NUnit.Framework;

namespace Test.Unit
{
  [TestFixture]
  public class TodoControllerTest
  {
    [Test]
    public void Should_Display_A_List_Of_Todo_Items()
    {

    }
  }
}
```

We have our test skeleton, similar to what we did in Section 2.2, *Test-Driving "Hello World"*, on page 38. Now we'll fill it with an assertion. The controller should display to-dos, so our assertion needs to verify that to-do items load. However, this will generate a couple of compiler errors, since neither a Todo class nor a TodoController exists. Let's work on creating these classes first before we return to this test, starting by creating a Todo model.

A model is a normal class. There are no special templates or wizards like there are for views and controllers. To create a new model, right-click the Solution Explorer, choose Add New Item, and select the Class template.

An alternative way to solve our compiler problem would be to generate the classes with ReSharper. While our mouse is over the compiler error

ReSharper Tip: Creating New Classes

While your mouse is over the Solution Explorer, hit Ctrl+Alt+Ins, and you'll be able to create and name a new class.

on Todo on line 5, we can use the ReSharper shortcut Alt+Enter to generate our missing Todo class (see the sidebar on page 59). This also works for controllers, but we don't get the generated template that MVC gives us.

For this new model, start by adding two properties, Title and Completed, and then add a default list of things to be done. This gives us a primitive way of saving our list. Static lists are *never* a good way to store information in real-world applications. We'll eventually replace the static lists in Chapter 8, *Persisting Your Models*, on page 174 when we introduce NHibernate.

Testing models is critical because they'll eventually hold important logic about how your system behaves. Since we're currently focusing on controller testing, let's deal with model testing a little later.

Download gettingorganized/Todo.cs

```
Line 1  namespace GetOrganized.Models
-       {
-         public class Todo
-         {
5          public static List<Todo> ThingsToBeDone = new List<Todo>
-          {
-            new Todo {Title = "Get Milk", Completed = false},
-            new Todo {Title = "Bring Home Bacon", Completed = false}
-          };
10
-          public bool Completed { get; set; }
-          public string Title { get; set; }
-        }
-      }
```

Our first model has a List<Todo> and a couple of *auto properties*. Auto setters are a new C# 3.0 feature to reduce the amount of code required to have simple getters and setters. Instead of writing out public bool Completed {get {return completed;} } and then having to create the private boolean field completed, the *auto setter property* is shorter, as shown on line 11.

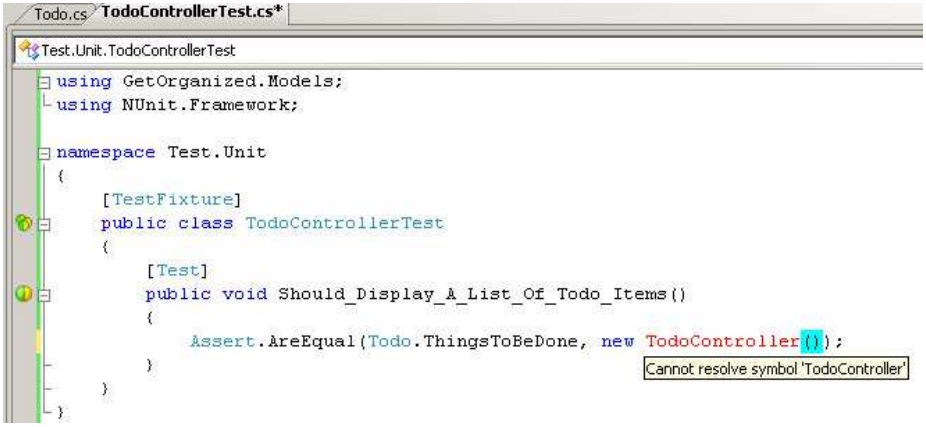

Figure 3.4: There will be classes that don't exist as you write your tests. This is a normal part of TDD.

With the model in place, we've removed one of the compiler errors. However, we're still getting another one because there is no such thing as TodoController (Figure 3.4). Not to worry, this is a regular part of practicing TDD. You'll find yourself regularly inventing new classes to satisfy what you're testing. Eventually, you'll get to the point of a compiling and failing test.

To remove the compiler error, we'll create the TodoController. Creating a controller involves right-clicking the Controller folder, selecting Add Controller, and inputting the name of the controller. Make sure to check the "Add action methods for Create, Update, Delete, and Details Scenarios" box, because we'll use these stubs later. The code generated for the TodoController looks like this:

Download gettingorganized/TodoController.cs

```
Line 1  namespace GetOrganized.Controllers
-       {
-         public class TodoController : Controller
-         {
5          //
-          // GET: /Todo/
-
-          public ActionResult Index()
-          {
10           return View();
-          }
-         }
-       }
```

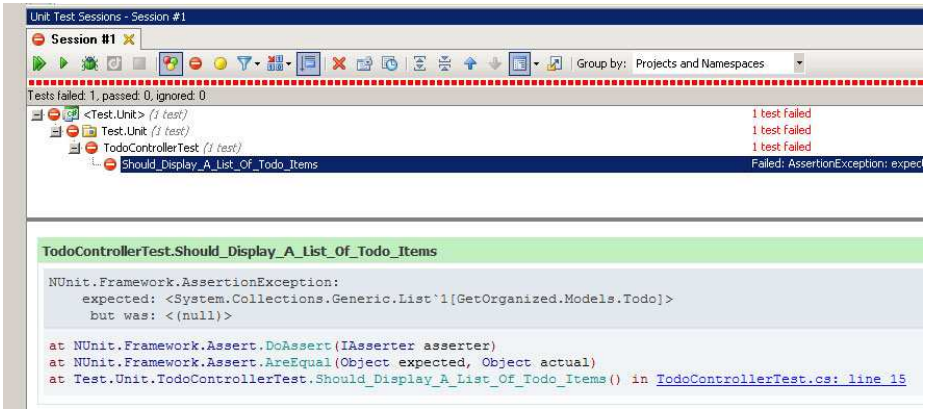

Figure 3.5: Should_Display_A_List_of_Todo_Items() is failing because the Index() action isn't meeting our expectations.

The helpful code comment on line 6 tells us that when we type in the URL http://localhost/Todo, we get the method that we're after. Note that Index() has a default route. The URL http://localhost/Todo/Index is equivalent to http://localhost/Todo, because this is specified in Global.asax.cs. Notice that the return value on the controller methods is an ActionResult object. Views use these objects for rendering purposes, but most important, they contain the model that we will attach to get this test to pass. ActionResults are covered in detail in Section 5.1, *Directing to Different Content Types with ActionResults*, on page 98.

To complete our assert statement, we'll need to compare apples to apples, or in this case to lists of Todo items. To achieve this, we need to cast ActionResult as a ViewResult object. The ViewResult class is a subtype of ActionResult that has a property called ViewData; this property is the key to passing the model between the controller and the view.

ViewData is a collection of objects. It has a special property called Model, which is where the model is set and accessed in the controller. We're expecting our controller to set our ViewData.Model to be our Todo list. For the code to compile, we'll need to add System.Web.Mvc to our references in the Test.Unit project. The code looks like this:

Download gettingorganized/TodoControllerTest.cs

```
Line 1  [Test]
2       public void Should_Display_A_List_Of_Todo_Items()
3       {
4           var viewResult = (ViewResult) new TodoController().Index() ;
5           Assert.AreEqual(Todo.ThingsToBeDone, viewResult.ViewData.Model );
6       }
```

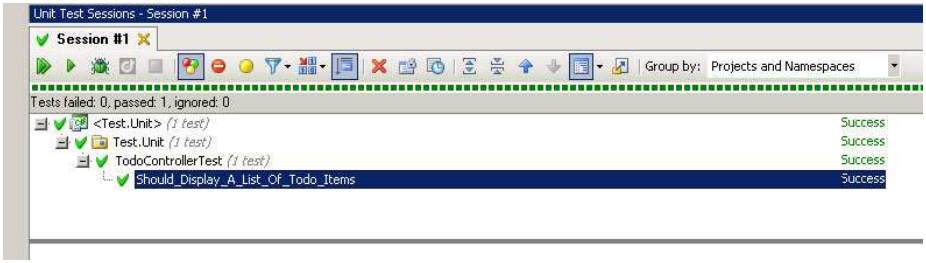

Figure 3.6: Adding the model to the controller makes our test pass.

Our code is compiling, and it's time to run the test and see whether it fails. Our comparison is failing when we run the test (Figure 3.5, on the preceding page). This means we've reached step 2 of the TDD cycle—"Watch the test fail" (Figure 2.2, on page 36). To reach step 3—"Get the test to pass"—we'll need to implement the Index() action to meet our assertion.

Currently our Index() action simply returns a ViewResult and therefore will fail. Let's wire up the model and get the test to pass (Figure 3.6):

Download gettingorganized/TodoController.cs

```
public class TodoController : Controller
{
  //
  // GET: /Todo/

  public ActionResult Index()
  {
    ViewData.Model = Todo.ThingsToBeDone;
    return View();
  }
}
```

Adding a View

Excellent, we've got our first passing test. But we still don't have anything the user can see. We need to add a view to complete the cycle. Adding a view is similar to the process of adding a controller. Simply right-click anywhere in the controller's action code, and select Add View (Figure 3.7, on page 56). We'll create a strongly typed view with the template called List to generate the HTML that lists the List<Todo> for us. The bottom of the dialog box is where we can specify the use of a master page, which is a layout template for the whole site (we'll cover

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Home Page for Test-Drive ASP.NET MVC
http://pragprog.com/titles/jmasp
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/jmasp.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	support@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com
Contact us:	1-800-699-PROG (+1 919 847 3884)