# Walkthrough: Using TDD with ASP.NET MVC

**.NET Framework 4** 8 out of 27 rated this helpful

This walkthrough shows you how to develop an ASP.NET MVC application in Visual Studio using the test-driven development (TDD) approach. MVC was designed to enable testability without requiring dependencies on a Web server (IIS), on a database, or on external classes. (This is in contrast to unit tests for Web Forms pages, which require a Web server.)

In this walkthrough, you will create tests for an MVC controller before you implement the controller functionality. You can write tests before you have a controller. The advantage is that compiler errors in your unit tests are then a first-level form of unit-test failure. The emphasis is on how to design the intent of the controller by writing unit tests before implementing the controller itself, which is an important aspect of the TDD philosophy. (For a good overview of the MVC TDD philosophy, see the entry It's Not TDD, It's Design By Example on Brad Wilson's blog.)

In the walkthrough, you will create a project and unit tests for an application that you can use to display and edit contacts. A contact has a first name, last name, phone number, and an email alias.

A Visual Studio project with source code is available to accompany this topic: Download. The download contains the completed ASP.NET MVC C# and Visual Basic projects (in the cs and vb folders) and C# and Visual Basic projects in the QuickStart folder. The first part of this walkthrough shows how to create an MVC project in Visual Studio, how to add a data model, and how to add metadata to the data model. If you know how to create an ASP.NET MVC project and an Entity Framework data model, you can use the projects that are in the downloadable project under the QuickStart folder instead and skip to the section Adding a Repository later in this walkthrough.

## Prerequisites

In order to complete this walkthrough, you will need:

- Microsoft Visual Studio 2008 Service Pack 1 or later.

  | Note |
  | --- |
  | Visual Studio Standard Edition and Visual Web Developer Express do not do not support unit-test projects. |

- The ASP.NET MVC 2 framework. To download the most up-to-date version of the framework, see the ASP.NET MVC download page.
- The Contact.mdf database file. This database file is part of the sample project that you can download for this project: Download

## Creating a New MVC Application with Unit Tests

# Creating a New MVC Application with Unit Tests

In this section, you create a new Visual Studio solution that includes both the application project and a test project.

| Note |
| --- |
| This walkthrough uses the Visual Studio unit testing framework. For information about how to add another unit-testing framework, see Walkthrough: Creating a Basic MVC Project with Unit Tests in Visual Studio |

### To create an MVC application with unit tests

1. In Visual Studio, in the **File** menu, click **New Project**.
2. In the **New Project** dialog box, under **Installed Templates**, open the **Visual C#** or **Visual Basic** node and then select **Web**.
3. Select the **ASP.NET MVC Web Application** template.
4. Name the solution **MvcContacts**.
5. Click **OK**.
6. When the **Create Unit Test Project** dialog box is displayed, make sure **Yes, create a unit test project** is selected, and then click **OK**.

   Visual Studio creates a solution that contains two projects, one named `MvcContacts` and one named `MvcContacts.Tests`.
7. On the **Test** menu, click **Run**, and then click **All Tests in Solution**.

   The results are displayed in the **Test Results** window. The tests pass.
8. In the `MvcContacts.Tests` project, open and examine the account controller test class (MvcContacts\MvcContacts.Tests\Controllers\AccountControllerTest) and the account controller model class (MvcContacts\Models\AccountModels).

   These classes provide a good introduction to how to create mock interfaces and to TDD. Mocking is the process of creating simple substitute (mock) objects for the dependencies in a class so you can test the class without the dependencies. To test interfaces, you typically create a mock class that implements the interface you want to test. For example, the `MockMembershipService` class in the account controller test class implements the **IMembershipService** interface to mock members that are part of membership classes, such as the `ValidateUser`, `CreateUser`, and `ChangePassword` methods. The `MockMembershipService` class enables you to test the action methods that create user accounts, validate user registration information, and change a user's password without having to instantiate a membership class like Membership.

# Creating a Database Model

This walkthrough uses an Entity Data model (EDM) that is created from the Contact database that is included in the downloadable sample project. (You must download the project in order to get the Contact.mdf file. For more information, see the Prerequisites section earlier in this walkthrough.)

### To create the database model

1. In **Solution Explorer**, right-click the App_Data folder of the `MvcContacts` project, click **Add**, and then click **Existing Item**.

   The **Add Existing Item** dialog box is displayed.

2. Navigate to the folder that contains the Contact.mdf file, select the Contact.mdf file, and then click **Add**.

3. In **Solution Explorer**, right-click the `MvcContacts` project, click **Add**, and then click **New Item**.

   The **Add New Item** dialog box is displayed.

4. Under **Installed Templates**, open the **Visual C#** or **Visual Basic** node, select **Data**, and then select the **ADO.NET Entity Data Model** template.

5. In the **Name** box enter, **ContactModel** and then click **Add**.

   The **Entity Data Model Wizard** window is displayed.

6. Under **What should the model contain**, select **Generate from database** and then click **Next**.

7. Under **Which data connection should your application use to connect to the database?**, select Contact.mdf.

8. Make sure that the **Save entity connection settings in Web.config as** check box is selected. You can leave the default connection string name.

9. Click **Next**.

   The wizard displays a page where you can specify what database objects you want to include in your model.

10. Select the **Tables** node to select the Contacts table. You can leave the default model namespace.

11. Click **Finish**.

    The ADO.NET Entity Data Model Designer is displayed. Close the designer.

## Adding Model Metadata

In this section, you will add the contact metadata. The contact class metadata will not be used in unit tests. However, it makes the sample more complete because it provides automated client-side and server-side data validation.

### To add model metadata

1. In the MvcContacts\Models folder, create a new class file named ContactMD.

   In this file, you'll add a class (`ContactMD`) that will contain the metadata for the `Contact` entity object that is part of the data model that you are using for this walkthrough.

2. Replace the code in the file with the following code:

C#

```
using System.ComponentModel.DataAnnotations;

namespace MvcContacts.Models {
    [MetadataType(typeof(ContactMD))]
    public partial class Contact {
        public class ContactMD {
            [ScaffoldColumn(false)]
            public object Id { get; set; }
            [Required()]
            public object FirstName { get; set; }
            [Required()]
            public object LastName { get; set; }
            [RegularExpression(@"^\d{3}-?\d{3}-?\d{4}$")]
            public object Phone { get; set; }
            [Required()]
            [DataType(DataType.EmailAddress)]
            [RegularExpression(@"^([0-9a-zA-Z]([-.\w]*[0-9a-zA-Z])*@([0-9a-
            public object Email { get; set; }
        }
    }
}
```

# Adding a Repository

A best practice in MVC is not to include Entity Data Model (EDM) or any other data-access framework code inside your controller. Instead, you should use the repository pattern. The repository sits between the application and the data store. A repository separates the business logic from interactions with the underlying data base and focuses your data access in one area, making it easier to create and maintain.

The repository returns objects from the domain model. For simple models, such as the model that you work with in this walkthrough, the objects returned from the EDM, LINQ to SQL, and other data models qualify as domain objects.

For more complex applications, a mapping layer might be required. A mapping layer is not necessarily inefficient. LINQ providers can create efficient queries to the back-end data store (that is, they can query using a minimal number of intermediate objects).

The repository should not require knowledge of EDM, LINQ to SQL, or any other data model you are working with. (Although LINQ is not covered in this walkthrough, using LINQ as the querying abstraction means you can hide the data storage mechanism. For example, this lets you use SQL Server for production and LINQ to Objects over in-memory collections for testing.)

Testing the action methods in a controller that directly access the EDM requires a connection to a database, because the action methods are dependent on the EDM (which is dependent on a database). The following code shows a MVC controller that uses the Contact entity of the EDM directly, and it

provides a simple example of why mixing database calls in action methods makes the action method difficult to test. For example, unit tests that edit and delete data change the state of the database. This requires each pass of the unit tests to have a clean database setup. Additionally, database calls are very expensive, whereas unit tests should be lightweight so they can be run frequently while you are developing your application.

**C#**

```
public class NotTDDController : Controller {

    ContactEntities _db = new ContactEntities();

    public ActionResult Index() {
        var dn = _db.Contacts;
        return View(dn);
    }

    public ActionResult Edit(int id) {
        Contact prd = _db.Contacts.FirstOrDefault(d => d.Id == id);
        return View(prd);
    }

    [HttpPost]
    public ActionResult Edit(int id, FormCollection collection) {
        Contact prd = _db.Contacts.FirstOrDefault(d => d.Id == id);
        UpdateModel(prd);
        _db.SaveChanges();
        return RedirectToAction("Index");
    }
}
```

The repository pattern has the following benefits:

- It provides a substitution point for the unit tests. You can easily test business logic without a database and other external dependencies.
- Duplicated queries and data-access patterns can be removed and refactored into the repository.
- Controller methods can use strongly typed parameters, which means the compiler finds data-typing errors each time you compile instead of relying of finding data-typing errors at run time when testing.
- Data access is centralized, which provides the following benefits:
-
    - Greater separation of concerns (SoC), another tenet of MVC, which increases maintainability and readability.
    - Simplified implementation of centralized data caching.
    - A more flexible and less coupled architecture that can be adapted as the overall design of the application evolves.
- Behavior can be associated with related data. For example, you can calculate fields or enforce

complex relationships or business rules between the data elements within an entity.

- A domain model can be applied in order to simplify complex business logic.

Using the repository pattern with MVC and TDD typically requires you to create an interface for your data-access class. The repository interface will make it easy to inject a mock repository when you unit test our controller methods.

In this section, you will add a contact repository, which is a class that is used to save the contacts in a database. You will also add an interface for the contact repository.

## To add a repository

1. In the MvcContacts\Models folder, create a class file and add a class named `IContactRepository`.

   The `IContactRepository` class will contain the interface for the repository object.

2. Replace the code in the class file with the following code:

   **C#**

```
using System;
using System.Collections.Generic;

namespace MvcContacts.Models {
    public interface IContactRepository {
        void CreateNewContact(Contact contactToCreate);
        void DeleteContact(int id);
        Contact GetContactByID(int id);
        IEnumerable<Contact> GetAllContacts();
        int SaveChanges();

    }
}
```

3. In the MvcContacts\Models folder, create a new class named `EntityContactManagerRepository`.

   The `EntityContactManagerRepository` class will implement the `IContactRepository` interface for the repository object.

4. Replace the code in the `EntityContactManagerRepository` class with the following code:

   **C#**

```
using System.Collections.Generic;
using System.Linq;

namespace MvcContacts.Models {
    public class EF_ContactRepository : MvcContacts.Models.IContactReposito
```

```
        private ContactEntities _db = new ContactEntities();

        public Contact GetContactByID(int id) {
            return _db.Contacts.FirstOrDefault(d => d.Id == id);
        }

        public IEnumerable<Contact> GetAllContacts() {
            return _db.Contacts.ToList();
        }

        public void CreateNewContact(Contact contactToCreate) {
            _db.AddToContacts(contactToCreate);
            _db.SaveChanges();
         //   return contactToCreate;
        }

        public int SaveChanges() {
            return _db.SaveChanges();
        }

        public void DeleteContact(int id) {
            var conToDel = GetContactByID(id);
            _db.Contacts.DeleteObject(conToDel);
            _db.SaveChanges();
        }

    }
}
```

# Creating Tests to Drive Design

In this section, you will add a mock implementation of the repository, add unit tests, and implement the application functionality from the unit tests.

## To implement the in-memory repository

1. In the `MvcContacts.Tests` project, create a Models folder.

2. In the MvcContacts.Tests\Models folder, create a new class named `InMemoryContactRepository`.

   The `InMemoryContactRepository` class will implement the `IContactRepository` interface that you created previously and have a simple repository for driving the application design.

3. Replace the code in the `InMemoryContactRepository` class with the following code:

**C#**

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;

using MvcContacts.Models;

namespace MvcContacts.Tests.Models {
    class InMemoryContactRepository : MvcContacts.Models.IContactRepository
        private List<Contact> _db = new List<Contact>();

        public Exception ExceptionToThrow { get; set; }
        //public List<Contact> Items { get; set; }

        public void SaveChanges(Contact contactToUpdate) {

            foreach (Contact contact in _db) {
                if (contact.Id == contactToUpdate.Id) {
                    _db.Remove(contact);
                    _db.Add(contactToUpdate);
                    break;
                }
            }
        }

        public void Add(Contact contactToAdd) {
            _db.Add(contactToAdd);
        }

        public Contact GetContactByID(int id) {
            return _db.FirstOrDefault(d => d.Id == id);
        }

        public void CreateNewContact(Contact contactToCreate) {
            if (ExceptionToThrow != null)
                throw ExceptionToThrow;

            _db.Add(contactToCreate);
           // return contactToCreate;
        }

        public int SaveChanges() {
            return 1;
        }

        public IEnumerable<Contact> GetAllContacts() {
            return _db.ToList();
        }


        public void DeleteContact(int id) {
            _db.Remove(GetContactByID(id));
        }

    }
}
```

## To add test support

1. In the MvcContacts project, open the Controllers\HomeController.cs file. Replace the code in the Controllers\HomeController.cs file with the following code:

```
using System;
using System.Web.Mvc;
using MvcContacts.Models;

namespace MvcContacts.Controllers {
    [HandleError]
    public class HomeController : Controller {
        IContactRepository _repository;
        public HomeController() : this(new EF_ContactRepository()) { }
        public HomeController(IContactRepository repository) {
            _repository = repository;
        }
        public ViewResult Index() {
            throw new NotImplementedException();
        }
    }
}
```

```
<HandleError()> _
Public Class HomeController
    Inherits System.Web.Mvc.Controller

    Private _repository As IContactRepository

    Public Sub New()
        Me.New(New EF_ContactRepository())
    End Sub

    Public Sub New(ByVal repository As IContactRepository)
        _repository = repository
    End Sub

    Public Function Index() As ViewResult
        Throw New NotImplementedException()
        Return View()
    End Function
End Class
```

This class contains two constructors. One is a parameterless constructor. The other takes a parameter of type IContactRepository; this constructor will be used by the unit tests to pass in the mock repository. The parameterless constructor creates an instance of the EF_ContactRepository class and is called by the MVC pipeline when an action method in the controller is invoked.

2. Close the HomeController file.

3. In the `MvcContacts.Test` project, open the Controllers\HomeControllerTest file and replace the code in the file with the following code:

```
using System.Web.Mvc;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using MvcContacts.Controllers;

using MvcContacts.Models;
using MvcContacts.Tests.Models;
using System.Web;
using System.Web.Routing;
using System.Security.Principal;

namespace MvcContacts.Tests.Controllers {
    [TestClass]
    public class HomeControllerTest {

        Contact GetContact() {
            return GetContact(1, "Janet", "Gates");
        }

        Contact GetContact(int id, string fName, string lName) {
            return new Contact
            {
                Id = id,
                FirstName = fName,
                LastName = lName,
                Phone = "710-555-0173",
                Email = "janet1@adventure-works.com"
            };
        }

        private static HomeController GetHomeController(IContactRepository
            HomeController controller = new HomeController(repository);

            controller.ControllerContext = new ControllerContext()
            {
                Controller = controller,
                RequestContext = new RequestContext(new MockHttpContext(),
            };
            return controller;
        }


        private class MockHttpContext : HttpContextBase {
            private readonly IPrincipal _user = new GenericPrincipal(
                     new GenericIdentity("someUser"), null /* roles */);

            public override IPrincipal User {
                get {
                    return _user;
                }
                set {
```

```
                set {
                    base.User = value;
                }
            }
        }
    }
}
```

```
Imports System
Imports System.Security
Imports System.Security.Principal
Imports System.Web
Imports System.Web.Mvc
Imports System.Web.Routing
Imports System.Web.Security
Imports Microsoft.VisualStudio.TestTools.UnitTesting
Imports MvcContacts

<TestClass()> Public Class HomeControllerTest




   <TestMethod()>
    Public Sub Index_Get_AsksForIndexView()
        ' Arrange
        Dim controller = GetHomeController(New InMemoryContactRepository())
        ' Act
        Dim result As ViewResult = controller.Index()
        ' Assert
        Assert.AreEqual("Index", result.ViewName)
    End Sub
    ' </snippet8>

    Private Function GetContactID_1() As Contact
        Return GetContactNamed(1, "Janet", "Gates")
    End Function

    Private Function GetContactNamed(ByVal id As Integer, ByVal fName As St
        Return New Contact With {.Id = id,
                                 .FirstName = fName,
                                 .LastName = lName,
                                 .Phone = "710-555-0173",
                                 .Email = "janet1@adventure-works.com"}

    End Function

    Private Shared Function GetHomeController(ByVal repository As IContactR
        Dim controller As New HomeController(repository)

        controller.ControllerContext = New ControllerContext() With
                                       {.Controller = controller,
                                         .RequestContext = New RequestConte
```

```
                                            New RouteData())}
        Return controller
    End Function

    Private Class MockHttpContext
        Inherits HttpContextBase

        Private ReadOnly _user As IPrincipal = New GenericPrincipal(New Gen

        Public Overrides Property User() As IPrincipal
            Get
                Return _user
            End Get
            Set(ByVal value As System.Security.Principal.IPrincipal)
                MyBase.User = value
            End Set
        End Property
    End Class

End Class
```

The code in the preceding example contains two method overloads that get a contact (`GetContact`) and a method to get the `HomeController` object. The unit tests will call methods in the `HomeController`. The code also contains a class to mock the **HttpContext** object. The `MockHttpContext` class used in this sample is a simplified version of the one found in the `AcountControllerTest` class file created when you made a new MVC project with unit tests.

# Adding Tests

One of the tenets of TDD with MVC is that each test should drive a specific requirement in an action method. The test should not verify database or other components (although these components should be tested in the data access unit tests and in integration testing). Another goal is that test names should be very descriptive; short general names like `Creat_Post_Test1` will make it more difficult to understand the test when you have hundreds of tests.

In this part of the walkthrough, you will assume the design calls for the default method of the Home controller to return a list of contacts. The default method for the controller is `Index`, so the first test will verify that the controller returns the index view. When you made changes to the `Index` method earlier in this walkthrough, you changed it to return a `ViewResult` object, not the more general `ActionResult` object. When you know a method will always return a `ViewResult` object, you can simplify the unit tests by returning a `ViewResult` object from the controller method. When you return a `ViewResult` object, the unit test does not have to cast the typical `ActionResult` object to a `ViewResult` object as part of the test, which results in simpler and more readable tests

## To add the first test

1. In the `HomeControllerTest` class, add a unit test named `Index_Get_AsksForIndexView` that verifies that the `Index` method returns a view named `Index`.

   The following example shows the completed unit test.

**C#**

```
[TestMethod]
public void Index_Get_AsksForIndexView() {
    // Arrange
    var controller = GetHomeController(new InMemoryContactRepository());
    // Act
    ViewResult result = controller.Index();
    // Assert
    Assert.AreEqual("Index", result.ViewName);
}
```

2. On the **Test** menu, click **Run**, and then click **All Tests in Solution**.

   The results are displayed in the **Test Results** window. As expected, the `Index_Get_AsksForIndexView` unit test fails.

3. Implement the following `Index` method of the `HomeController` class in order to return a list of all the contacts.

```
public ViewResult Index() {
            return View("Index", _repository.ListContacts());
        }
```

   In the spirit of TDD, you write just enough code to satisfy the design test.

4. Run the tests. This time the `Index_Get_AsksForIndexView` unit test passes.

# Creating a Test for Retrieving Contacts

In this section you will verify that you can retrieve all the contacts. You do not want to create unit tests that test data access. Verifying your application can access the database and retrieve contacts is important, but that is an integration test, not a TDD unit test.

## To add a test for retrieving contacts

- Create a test that adds two contacts to the in-memory repository in the `HomeControllerTest` class and then verifies that they are contained in the ViewData Model object that is contained in the `Index` view.

  The following example shows the completed test.

**C#**

```
[TestMethod]
public void Index_Get_RetrievesAllContactsFromRepository() {
    // Arrange
    Contact contact1 = GetContactNamed(1, "Orlando", "Gee");
    Contact contact2 = GetContactNamed(2, "Keith", "Harris");
    InMemoryContactRepository repository = new InMemoryContactRepository();
    repository.Add(contact1);
    repository.Add(contact2);
    var controller = GetHomeController(repository);

    // Act
    var result = controller.Index();

    // Assert
    var model = (IEnumerable<Contact>)result.ViewData.Model;
    CollectionAssert.Contains(model.ToList(), contact1);
    CollectionAssert.Contains(model.ToList(), contact1);
}
```

# Creating a Test for Creating a Contact

You will now test the process of creating a new contact. The first test verifies that an HTTP POST operation has completed successfully and invoked the `Create` method using data that deliberately contains model errors. The result will not add a new contact but instead returns the HTTP GET `Create` view that contains the fields you have entered and the model errors. Running a unit test on a controller does not execute the MVC pipeline or model-binding process. Therefore, the model error would not be caught during the binding process. To account for this, the test adds a mock error.

## To add a test for creating a contact

1. Add the following test to the project:

**C#**

```
[TestMethod]
public void Create_Post_ReturnsViewIfModelStateIsNotValid() {
    // Arrange
    HomeController controller = GetHomeController(new InMemoryContactReposi
    // Simply executing a method during a unit test does just that - execut
    // The MVC pipeline doesn't run, so binding and validation don't run.
    controller.ModelState.AddModelError("", "mock error message");
    Contact model = GetContactNamed(1, "", "");

    // Act
    var result = (ViewResult)controller.Create(model);

    // Assert
    Assert.AreEqual("Create", result.ViewName);
}
```

The code shows how an attempt to add a contact that contains model errors returns the HTTP GET `Create` view.

2. Add the following test:

**C#**

```
[TestMethod]
public void Create_Post_PutsValidContactIntoRepository() {
    // Arrange
    InMemoryContactRepository repository = new InMemoryContactRepository();
    HomeController controller = GetHomeController(repository);
    Contact contact = GetContactID_1();

    // Act
    controller.Create(contact);

    // Assert
    IEnumerable<Contact> contacts = repository.GetAllContacts();
    Assert.IsTrue(contacts.Contains(contact));
}
```

The code shows how to verify that an HTTP POST to the `Create` method adds a valid contact to the repository.

A test that is often overlooked is to verify that methods correctly handle exceptions. The `InMemoryContactRepository` class lets you set a mock exception, simulating an exception that a database would raise when a constraint or other violation occurred. Many database exceptions cannot be caught with model validation, so it is important to verify exception handling code works correctly. The following example shows how to do this.

**C#**

```
[TestMethod]
public void Create_Post_ReturnsViewIfRepositoryThrowsException() {
    // Arrange
    InMemoryContactRepository repository = new InMemoryContactRepository();
    Exception exception = new Exception();
    repository.ExceptionToThrow = exception;
    HomeController controller = GetHomeController(repository);
    Contact model = GetContactID_1();

    // Act
    var result = (ViewResult)controller.Create(model);

    // Assert
```

```
        Assert.AreEqual("Create", result.ViewName);
        ModelState modelState = result.ViewData.ModelState[""];
        Assert.IsNotNull(modelState);
        Assert.IsTrue(modelState.Errors.Any());
        Assert.AreEqual(exception, modelState.Errors[0].Exception);
    }
```

## Next Steps

The downloadable sample includes more tests that are not detailed here. To learn more about how to use mock objects and how to use TDD methodology with MVC projects, review the additional tests, and write tests for the `Delete` and `Edit` methods.

## See Also

Tasks
How to: Add a Custom ASP.NET MVC Test Framework in Visual Studio
Other Resources
Unit Testing in ASP.NET MVC Applications
Testability and Entity Framework 4.0
Using Mocks And Tests To Design Role-Based Objects
Testing Routing and URL Generation in ASP.NET MVC

## Community Additions

### Good article but a design is not test driven if the design is done first...

This is a good article on how to test your code, and does talk about how TDD should drive a specific requirement but it doesn't seem to follow it's own advise.

I would have expected the first line of code to be a failing test in an empty mvc web application with no pre-existing code. Not using any IOC to start. The first bit of code could have been a simple version of the index test "public void Index_Get_AsksForIndexView() {}" without the initial repository or any other class.

The classes should be created to make the code compile, then build on top off to get the tests passing. Smallest most simple changes to get the test green. Then write another test, make them pass and repeat.

Once you got all the relevant tests passing for that controller or a specific behaviour you start looking at re-factoring using IOC as required and similar as you now have tests which will fail as soon as you break the expectations during re-factoring or start introducing abstraction which needs to be now included in the tests.


Gontronix
6/28/2013

## Very helpful but some thing missing

1) some code are missing: GetContacedNamed() GetContactID_D()

2)Where is download link? the Contact.mdf download link is a dead link. It will be very helpful to provide download for whole solution.

Yes it is anoying that the link is dead. Finally I realised the .mdf file is inside the solution itself. Wasted time: 5 minutes


danito_msdn
8/30/2012

## This isn't Test Driven Development

While the article has some good elements here it's missing the core of TDD, namely shaping your application and domain using tests. Starting with a database and controllers and views just means you've already committed yourself to a design and thus, you're not doing Test *Driven* Design anymore, you're just writing tests for already working code. This is a key element about TDD, not having tests but shaping the domain from them. This article misses the essence entirely.


Bil Simser [MVP]
5/12/2012

## Obviously a new way of doing "TDD"

*Öhhhh? TDD?*

This is NOT TDD:
*Creating a Database Model*
*Adding Model Metadata*
*Adding a Repository*
*Add Test support*
*->Add first test*


Thomas Lee
11/9/2011