Entity Framework in the Enterprise

You can download the complete code for all parts of this tutorial series here.

The complete series of articles for this tutorial includes:

1. Contoso University – A quick review of the Microsoft sample application and its enterprise ready features.
2. Unit Tests – Adding unit tests to Contoso to prove our changes don’t break the existing system.
3. SQL Server Database Project – Switching from migrations to a more flexible database project.
4. Compiled Views and Reverse Engineering – How to handle a large database with many tables.
5. Audit Tracking – Recording changes made to the database without changing the model objects.
6. Custom Security – Integrating into an existing corporate database for security.
7. Performance – Record, test and improve performance and scale in your application.
8. Business Rules – Creating a layered architecture to encapsulate business rules.
9. Entity Framework 6 – What’s changing in the newest version of Entity Framework.
10. What’s Left – Health Monitoring, Web API and oData, EF6

# Part 0 - Contoso University

In this tutorial I am going to show how you can take a sample MVC site built around Entity Framework and extend it to meet the challenges of an enterprise ready solution for the corporate environment. We will start with the excellent Contoso University application, but you can easily apply all the changes we are going to make to any Entity Framework base application. Note that these guidelines are based around Code First, but most of it still applies and could be adapted to the Database First or Model First approach.

## About Contoso University

If you are not familiar with the Contoso University sample application, it is detailed in a 10 part tutorial on the ASP.NET web site under [Getting Started with EF using MVC](http://www.asp.net/mvc/tutorials/getting-started-with-ef-using-mvc). I highly suggest you read these articles and work through them, even if you already know and use Entity Framework. There are some nice scenarios it covers that you may not be aware of yet.

You can download the completed tutorial code for Contoso Univerity, [ASP.NET MVC Application Using Entity Framework Code First](http://code.msdn.microsoft.com/ASPNET-MVC-Application-b01a9fe8) , at Microsoft’s Code Gallery. This is the code base we will be starting with, so it is also included in the complete code download for this tutorial series.

Let’s get started by discussing some of the features already built into Contoso University that you would likely need in an enterprise ready application.

## Sorting, Filtering and Paging

Any application that has more than a very small set of data is going to need these features added to keep the performance from your application from tumbling down. Part 3 covers adding a strategy for sorting and filtering, and handles paging by adding PagedList Nuget package.

## Concurrency

Handling two users changing data at the same time in a company is critical. A last in wins approach is usually not going to cut it. Part 7 shows how concurrency is easily built into Entity Framework using a timestamp column.

## Inheritance

Often in an company application you won’t have complete control of the database and might run into one of the table design patterns that are handled in Entity Framework by one of the inheritance patterns. Part 8 covers the differences between the Table-per-Heirarchy and the Table-per-Type patterns.

## Repository and Unit of Work

Anything beyond a basic MVC application where the controller does all the work will need some separation of duty between the business logic and data access. You will often find the repository and unit of work patterns in play in an application and Part 9 explains these concepts. Later in this tutorial we will extend this out further to make a clear separation of work between user interface and business logic using a façade pattern.

## Raw SQL Queries

Companies have DBAs and DBAs write stored procedures and these stored procedures don’t always fit into your model well for various reasons. Business applications interact with other business applications and it’s not always through a nice service layer, sometimes it’s just a data pump. For both of these scenarios, and more, it might just be easier to execute a SQL statement. Part 10 cover how to do this.

# Part 1 – Unit Tests

We’re going to make changes to the application, which means we might break something. So the first thing we are going to do is create some unit tests. These are by no means exhaustive unit tests, or indicative of the “best” way to automate the testing of your application. I want to focus more on the changes to Entity Framework. So the test we are creating will execute the controller actions to give us pretty good coverage of the functionality in the system.

To setup a new test project for an Entity Framework based MVC app, I typically do the following:

1. Start by adding a new Unit Test Project to the solution called Unit Tests.
2. Enable Code Analysis.
3. Open the App.config file and add the connection strings from the Web.config in the web project.
4. Open the Package Manager Console (Tools > Library Package Manager > Package Manager Console) and run update-database. This should create the database, apply the code migrations and seed the database with test data. For Contoso Univeristy, at this point in the process you might get an error “Cannot find the object “dbo.Student” because it does not exist or you do not have permissions. “ I ran into this because I ran the web before running the update database command. The easiest way to fix this is delete the database.
5. Run the web site.
6. Add a reference to the web project.
7. Add the EntityFramework, MVC and Moq NuGet packages to the project.
8. Add references to the Microsoft.CSharp and System.Web libraries.
9. Change the Assembly name and Default namespace to something more appropriate like ContosoUniveristy.UnitTests.
10. Update the AssemblyInfo.cs in the web project to make internals available to the unit test project. Ex: [assembly: InternalsVisibleTo("ContosoUniversity.UnitTests")]

There is one last step I want to go into more detail on, the creation of a static Helper class. We are going to run our unit tests against the seed data, so it’s important that before each unit test we reset the state of the data back to the same starting point. I will create a DataInitialize method that deletes all the records and reseeds the identity columns. In practice I have found this to execute faster than recreating the database from migration scripts that need to create the database structure all over again. You could keep a test database that you make a copy of, but I want to utilize migration scripts to apply test data and changes without keep a second copy in sync. This method will look something like this:

public static void DataInitialize()

{

    var context = new ContosoUniversity.DAL.SchoolContext();

    context.Database.ExecuteSqlCommand("DELETE FROM dbo.Enrollment; DBCC CHECKIDENT('dbo.Enrollment', RESEED, 0);");

    context.Database.ExecuteSqlCommand("DELETE FROM dbo.OfficeAssignment");

    context.Database.ExecuteSqlCommand("DELETE FROM dbo.CourseInstructor");

    context.Database.ExecuteSqlCommand("DELETE FROM dbo.Course");

    context.Database.ExecuteSqlCommand("DELETE FROM dbo.Department; DBCC CHECKIDENT('dbo.Department', RESEED, 0);");

    context.Database.ExecuteSqlCommand("DELETE FROM dbo.Person; DBCC CHECKIDENT('dbo.Person', RESEED, 0);");

    context.SaveChanges();

    var configuration = new ContosoUniversity.Migrations.Configuration();

    var migrator = new DbMigrator(configuration);

    migrator.Update();

}

This is by no means the only way to approach unit testing. Many people will push for a mock framework that tests the code without hitting the database at all. In fact, this would be a more “pure” approach to unit testing by reducing dependencies. But you should still be writing some tests to test the database.

There is one more thing we are going to add to the Helper class and that’s some mock helpers for form collection posts. ContosoUniversity passes in a form collection to the Instructor controller edit action. This is a common pattern in MVC. In order to unit test that, you need to mock up a fake form collection. This testing method come from the post [ASP.NET MVC Session at Mix08, TDD and MvcMockHelpers by Scott Hanselman](http://www.hanselman.com/blog/ASPNETMVCSessionAtMix08TDDAndMvcMockHelpers.aspx). I am not going to walk through that code, you can read Scott’s post to get the details. Here is an example of a unit test putting this in practice. The third line of code adds the fake controller context to the controller.

[TestMethod]

public void EditInstructor()

{

    var context = new SchoolContext();

    var controller = new InstructorController();

    controller.SetFakeControllerContext();

    var instructorId = 9;

    var expectedLastName = "Unit Test";

    var expectedCourseCount = 1;

    var selectedCourses = new string[] { "2021" };

    var formCollection = new FormCollection();

    formCollection.Add("LastName", expectedLastName);

    formCollection.Add("FirstMidName", "Kim");

    formCollection.Add("HireDate", "3/11/1995");

    formCollection.Add("OfficeAssignment.Location", "");

    controller.ValueProvider = formCollection.ToValueProvider();

    controller.Edit(instructorId, formCollection, selectedCourses);

    context = new SchoolContext();

    Assert.AreEqual(expectedLastName, context.Instructors.Find(instructorId).LastName);

    Assert.AreEqual(expectedCourseCount, context.Instructors.Find(instructorId).Courses.Count);

}

At this point I would recommend downloading the source code for this tutorial to see the Helper class and all the unit test in action. Make sure you can get them all to run before moving on to the next tutorial in the series.

More Information:

# Part 2 – SQL Server Database Project

When you are working in a corporate environment you will not always have complete control over your database. It’s quite common to work out of a shared database that multiple applications use, or other people may make changes without your knowledge. For example, a DBA could audit your database and make some “helpful” changes without you knowing until you find out the hard way when your migration script runs. Even if you have complete control over your database, you may have members of your team, such as database or report developers, that are not comfortable with C#.

The SQL Server Database Project type goes a long way toward solving these problems by giving you the ability to track your entire database source as SQL files and helping you create migration scripts for deploying changes.

Before you begin, make sure you have downloaded the [Microsoft SQL Server Data Tools](http://msdn.microsoft.com/en-us/data/tools.aspx).

1. Create a new SQL Server Database Project found under the SQL Server Templates.
2. Open up the Project properties, go to Build settings, and change the name of the output file to something more appropriate like EFEnterprise.
3. Still under Properties, go to the Debug settings, and edit the connection string to be:   
   Data Source=(LocalDb)\v11.0;Initial Catalog=EFEnterprise;Integrated Security=True;Pooling=False
4. Still under Properties, go to the Code Analysis setting, and enable Code Analysis on build.

Now you can import the existing database by right clicking on the project and selecting Import > Database. This will populate a dbo\Tables folder with one SQL file for each table in the database. Our database only has tables in it, so this is all you get.

We are going to add a Script folder with two scripts in it to replace the Seed method in the current migration. Create the Scripts folder, and add a new item of type Script, which should be directly under the Add menu when you right click. The first script is Reset.sql that will clean up the database and reset identity seeds.

-- Clean up order should be first in, last out.

DELETE FROM dbo.Enrollment; DBCC CHECKIDENT('dbo.Enrollment', RESEED, 0);

DELETE FROM dbo.OfficeAssignment;

DELETE FROM dbo.CourseInstructor;

DELETE FROM dbo.Course;

DELETE FROM dbo.Department; DBCC CHECKIDENT('dbo.Department', RESEED, 0);

DELETE FROM dbo.Person; DBCC CHECKIDENT('dbo.Person', RESEED, 0);

The second script is TestData.sql that will populate the database with the same data that the Seed method did.

-- Students

INSERT INTO dbo.Person (LastName, FirstName, EnrollmentDate, Discriminator) VALUES ('Alexander', 'Carson', '9/1/2010', 'Student'); DECLARE @Person\_Alexander INT = @@IDENTITY;

INSERT INTO dbo.Person (LastName, FirstName, EnrollmentDate, Discriminator) VALUES ('Alonso', 'Meredith', '9/1/2012', 'Student'); DECLARE @Person\_Alonso INT = @@IDENTITY;

INSERT INTO dbo.Person (LastName, FirstName, EnrollmentDate, Discriminator) VALUES ('Anand', 'Arturo', '9/1/2013', 'Student'); DECLARE @Person\_Anand INT = @@IDENTITY;

INSERT INTO dbo.Person (LastName, FirstName, EnrollmentDate, Discriminator) VALUES ('Barzdukas', 'Gytis', '9/1/2012', 'Student'); DECLARE @Person\_Barzdukas INT = @@IDENTITY;

INSERT INTO dbo.Person (LastName, FirstName, EnrollmentDate, Discriminator) VALUES ('Li', 'Yan', '9/1/2012', 'Student'); DECLARE @Person\_Li INT = @@IDENTITY;

INSERT INTO dbo.Person (LastName, FirstName, EnrollmentDate, Discriminator) VALUES ('Justice', 'Peggy', '9/1/2011', 'Student'); DECLARE @Person\_Justice INT = @@IDENTITY;

INSERT INTO dbo.Person (LastName, FirstName, EnrollmentDate, Discriminator) VALUES ('Norman', 'Laura', '9/1/2013', 'Student'); DECLARE @Person\_Norman INT = @@IDENTITY;

INSERT INTO dbo.Person (LastName, FirstName, EnrollmentDate, Discriminator) VALUES ('Olivetto', 'Nino', '9/1/2005', 'Student'); DECLARE @Person\_Olivetto INT = @@IDENTITY;

-- Instructors

INSERT INTO dbo.Person (LastName, FirstName, HireDate, Discriminator) VALUES ('Abercrombie', 'Kim', '3/11/1995', 'Instructor'); DECLARE @Person\_Abercrombie INT = @@IDENTITY;

INSERT INTO dbo.Person (LastName, FirstName, HireDate, Discriminator) VALUES ('Fakhouri', 'Fadi', '7/6/2002', 'Instructor'); DECLARE @Person\_Fakhouri INT = @@IDENTITY;

INSERT INTO dbo.Person (LastName, FirstName, HireDate, Discriminator) VALUES ('Harui', 'Roger', '7/1/1998', 'Instructor'); DECLARE @Person\_Harui INT = @@IDENTITY;

INSERT INTO dbo.Person (LastName, FirstName, HireDate, Discriminator) VALUES ('Kapoor', 'Candace', '1/15/2001', 'Instructor'); DECLARE @Person\_Kapoor INT = @@IDENTITY;

INSERT INTO dbo.Person (LastName, FirstName, HireDate, Discriminator) VALUES ('Zheng', 'Roger', '2/12/2004', 'Instructor'); DECLARE @Person\_Zheng INT = @@IDENTITY;

-- Departments

INSERT INTO dbo.Department (Name, Budget, StartDate, PersonID) VALUES ('English', 350000, '9/1/2007', @Person\_Abercrombie); DECLARE @Department\_English INT = @@IDENTITY;

INSERT INTO dbo.Department (Name, Budget, StartDate, PersonID) VALUES ('Mathematics', 100000, '9/1/2007', @Person\_Fakhouri); DECLARE @Department\_Mathematics INT = @@IDENTITY;

INSERT INTO dbo.Department (Name, Budget, StartDate, PersonID) VALUES ('Engineering', 350000, '9/1/2007', @Person\_Harui); DECLARE @Department\_Engineering INT = @@IDENTITY;

INSERT INTO dbo.Department (Name, Budget, StartDate, PersonID) VALUES ('Economics', 100000, '9/1/2007', @Person\_Kapoor); DECLARE @Department\_Economics INT = @@IDENTITY;

-- Courses

DECLARE @Course\_Chemistry INT = 1050; INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID) VALUES (@Course\_Chemistry, 'Chemistry', 3, @Department\_Engineering);

DECLARE @Course\_Microeconomics INT = 4022; INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID) VALUES (@Course\_Microeconomics, 'Microeconomics', 3, @Department\_Economics);

DECLARE @Course\_Macroeconomics INT = 4041; INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID) VALUES (@Course\_Macroeconomics, 'Macroeconomics', 3, @Department\_Economics);

DECLARE @Course\_Calculus INT = 1045; INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID) VALUES (@Course\_Calculus, 'Calculus', 4, @Department\_Mathematics);

DECLARE @Course\_Trigonometry INT = 3141; INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID) VALUES (@Course\_Trigonometry, 'Trigonometry', 4, @Department\_Mathematics);

DECLARE @Course\_Composition INT = 2021; INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID) VALUES (@Course\_Composition, 'Composition', 3, @Department\_English);

DECLARE @Course\_Literature INT = 2042; INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID) VALUES (@Course\_Literature, 'Literature', 4, @Department\_English);

-- Course Instructors

INSERT INTO dbo.CourseInstructor(CourseID, PersonID) VALUES (@Course\_Composition, @Person\_Abercrombie);

INSERT INTO dbo.CourseInstructor(CourseID, PersonID) VALUES (@Course\_Literature, @Person\_Abercrombie);

INSERT INTO dbo.CourseInstructor(CourseID, PersonID) VALUES (@Course\_Chemistry, @Person\_Harui);

INSERT INTO dbo.CourseInstructor(CourseID, PersonID) VALUES (@Course\_Trigonometry, @Person\_Harui);

INSERT INTO dbo.CourseInstructor(CourseID, PersonID) VALUES (@Course\_Chemistry, @Person\_Kapoor);

INSERT INTO dbo.CourseInstructor(CourseID, PersonID) VALUES (@Course\_Microeconomics, @Person\_Zheng);

INSERT INTO dbo.CourseInstructor(CourseID, PersonID) VALUES (@Course\_Macroeconomics, @Person\_Zheng);

-- Office Assignment

INSERT INTO dbo.OfficeAssignment (PersonID, Location) VALUES (@Person\_Fakhouri, 'Smith 17');

INSERT INTO dbo.OfficeAssignment (PersonID, Location) VALUES (@Person\_Harui, 'Gowan 27');

INSERT INTO dbo.OfficeAssignment (PersonID, Location) VALUES (@Person\_Kapoor, 'Thompson 304');

-- Enrollments

DECLARE @Enrollment\_Grade\_A INT = 0;

DECLARE @Enrollment\_Grade\_B INT = 1;

DECLARE @Enrollment\_Grade\_C INT = 2;

DECLARE @Enrollment\_Grade\_D INT = 3;

DECLARE @Enrollment\_Grade\_F INT = 4;

INSERT INTO dbo.Enrollment (CourseID, PersonID, Grade) VALUES (@Course\_Chemistry, @Person\_Alexander, @Enrollment\_Grade\_A);

INSERT INTO dbo.Enrollment (CourseID, PersonID, Grade) VALUES (@Course\_Microeconomics, @Person\_Alexander, @Enrollment\_Grade\_C);

INSERT INTO dbo.Enrollment (CourseID, PersonID, Grade) VALUES (@Course\_Macroeconomics, @Person\_Alexander, @Enrollment\_Grade\_B);

INSERT INTO dbo.Enrollment (CourseID, PersonID, Grade) VALUES (@Course\_Trigonometry, @Person\_Alonso, @Enrollment\_Grade\_B);

INSERT INTO dbo.Enrollment (CourseID, PersonID, Grade) VALUES (@Course\_Trigonometry, @Person\_Alonso, @Enrollment\_Grade\_B);

INSERT INTO dbo.Enrollment (CourseID, PersonID, Grade) VALUES (@Course\_Composition, @Person\_Alonso, @Enrollment\_Grade\_B);

INSERT INTO dbo.Enrollment (CourseID, PersonID, Grade) VALUES (@Course\_Chemistry, @Person\_Anand, NULL);

INSERT INTO dbo.Enrollment (CourseID, PersonID, Grade) VALUES (@Course\_Microeconomics, @Person\_Anand, @Enrollment\_Grade\_B);

INSERT INTO dbo.Enrollment (CourseID, PersonID, Grade) VALUES (@Course\_Chemistry, @Person\_Barzdukas, @Enrollment\_Grade\_B);

INSERT INTO dbo.Enrollment (CourseID, PersonID, Grade) VALUES (@Course\_Composition, @Person\_Li, @Enrollment\_Grade\_B);

INSERT INTO dbo.Enrollment (CourseID, PersonID, Grade) VALUES (@Course\_Literature, @Person\_Justice, @Enrollment\_Grade\_B);

Next, add a new file to the project of type Post-Deployment Script under User Scripts, named Script.PostDeployment.sql. This is the only one we will be adding, but you’ll also notice scripts for pre-deployment, scripts to include in the build, or standalone script files you can run manually. This script will run whenever you deploy the database and it is responsible for kicking off the two other files.

/\*

These are in the opposite order that they are listed in Clean.sql

The test project should run these in the same order listed here.

See UnitTests.Helper.

\*/

PRINT 'Deleting existing test data.';

:r Scripts\Reset.sql

PRINT 'Populate test data.';

:r Scripts\TestData.sql

Next we are going to update the Helper class in the test project to also run the scripts. The approach I am taking is to recreate the launching of the individual script files, not run the post deployment script. That would require you run the SQLCMD executable, which is possible but adds a couple of seconds to each test. You could combine all three of these scripts into a single post deployment script and that would make your life easier in the short run, but as you add more test data you will want to start breaking out your scripts into smaller more manageable sizes. It’s quite possible the two separate scripts we have created will not be enough for your project.

public static void DataInitialize()

{

    Database.SetInitializer<SchoolContext>(null);

    var scripts = new List<string>();

    var postDeploy = File.ReadAllLines(@"..\..\..\Database\Script.PostDeployment.sql");

    foreach (var line in postDeploy)

    {

        var start = line.IndexOf(@":r Scripts\");

        if (start >= 0)

        {

            start = start + 11;

            scripts.Add(File.ReadAllText(@"..\..\..\Database\Scripts\" + line.Substring(start, line.Length - start)));

        }

    }

    using (var cn = new SqlConnection(ConfigurationManager.ConnectionStrings["SchoolContext"].ConnectionString))

    {

        cn.Open();

        foreach (var script in scripts)

            using (var cmd = new SqlCommand(script, cn))

                cmd.ExecuteNonQuery();

    }

}

When you build the test project you are going to get a Code Analysis warning: “CA2100:Review SQL queries for security vulnerabilities”. This is because we are reading SQL from a file and executing it, which could be dangerous if someone changes the file. Since we are only going to run this against a database it’s okay to suppress this message.

You may have noticed in the helper class we also turned off data migration. You’ll need to do the same thing in the Application\_Start method of the Global.asax file in the web project.

protected void Application\_Start()

{

    AreaRegistration.RegisterAllAreas();

    WebApiConfig.Register(GlobalConfiguration.Configuration);

    FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

    RouteConfig.RegisterRoutes(RouteTable.Routes);

    BundleConfig.RegisterBundles(BundleTable.Bundles);

    AuthConfig.RegisterAuth();

    Database.SetInitializer<SchoolContext>(null);

}

Finally, for Contoso University, we need to update the Contact method in the HomeController to comment out the migration code which was setup as a manual test.

// Click the DbgSeed link to debug the Seed method

public ActionResult Contact()

{

    //var configuration = new Configuration();

    //var migrator = new DbMigrator(configuration);

    //migrator.Update();

    ViewBag.Message = "Your contact page.";

    return View();

}

You are now ready to test your changes. Start by deleting the existing database and running the database project in Debug mode, this is an easy way to deploy your changes without setting up a publish file. The database should be recreated and populated with test data. Next, run all your unit tests again to make sure everything is still working.

Before running the web project, update the connection strings to remove the AttachDBFilename parameter. Now run the website and make sure it is also working.

You now have a database project setup and working that has replaced the code migration features in Entity Framework. There is one last thing we want to do. In a real corporate environment, it is highly unlikely that you will deploy your changes by pointing Visual Studio at the live database and running the database project. Instead you can create a migration script by right clicking on the database project and selecting Schema Compare. Set the project as the source on the left and your actual database as the target on the right. You could update the database immediately after the compare, but you can also save it as a script for deployment by a DBA or include it in an install package.

More Info:

There is a lot more to the SQL Server database project and the Microsoft SQL Server Data Tools than I have covered here. Be sure to check out its full feature set on the MSDN site and get an overview here.

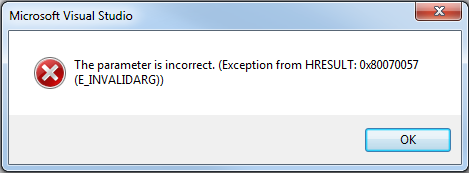
# Part 3 – Compiled Views and Reverse Engineering

In the case of Contoso University, the database was created along with the application. But in many scenarios, the database will already exist. Microsoft has a set of [Entity Framework Power Tools](http://visualstudiogallery.msdn.microsoft.com/72a60b14-1581-4b9b-89f2-846072eff19d) that you can download to help reverse engineer an existing database. Any code that is reverse engineered is subject to developer preferences and may not look exactly the way you would have written it. For example, the code generated by this tool makes use of the Fluent API instead of data annotations. It’s quite possible to add the data annotations yourself and remove the Fluent API if that is your preference or you can customize the templates to generate the code how you want. But the code generator does a good job getting you up and running quickly.

The other thing provided by the Power Tools is the ability to generate pre-compiled views. Why is this important? What the Entity Framework does is compile the SQL needed to talk to your database on demand. The first time you make a request to execute against the database this will happen. The results stay in memory for the life of the running application. This will normally cause a very slight performance hit and not an issue. But when your database becomes very large and complex, this time can be unreasonable. On one of my projects that include hundreds of tables, with sometimes complex relationships, this process will take about 3 minutes. One way of getting around this for an ASP.NET application is to include something in the application warm up to avoid the first user from taking the hit. But a better way is to compile the views into the source code. This option will also greatly improve the speed of the unit tests.

Right click on the SchoolContext in the DAL folder of the Contoso University project and select Entity Framework > Generate Views. This will create a new class called SchoolContext.Views. If you open this file you will be able to see the SQL that is being generated.

*Note: Hey it won’t generate. I’m getting some crazy unhelpful error “The parameter is incorrect.”*

**

*The power tools are beta code and sometimes they behave like beta code. One problem I have found when trying to generate views from a web project is that most unhelpful error message. You have to make sure that project containing the Context class you are using is also marked as the Startup project. If you get other error messages, remember that* [*Stack Overflow*](http://stackoverflow.com/) *is your friend.*

There is a downside to this that probably will not affect you in a corporate environment. This SQL is tailored to the database you are connecting to, so this code is not portable between different databases like SQL Server and Oracle. So you are giving up some portability for speed. This is not likely to affect you in a corporate environment like it would if you were packaging software for customer use.

Finally we are going to do one more thing to improve performance, and that is to get the application to cache the views. The following lines of code should be added to the unit test Helper class

// Execute a context to cache views, so performance tests are real world.

new ContosoUniversity.DAL.CourseRepository(new SchoolContext()).Get().First();

and the Global.asax file in the web project.

public class PreWarmCache : IProcessHostPreloadClient

{

    public void Preload(string[] parameters)

    {

        // Execute a model read to load views.

        new ContosoUniversity.DAL.CourseRepository(new SchoolContext()).Get().First();

    }

}

Make sure you run the website and all your unit test again to make sure everything is still working.

More Information:

# Part 4 – Audit Tracking

It’s often a requirement, whether by your organization or legal regulations, that you track changes to the database. This could be done to log files, but more often needs to be written to the database for easier reporting or application integration. Let’s look at some options on how to accomplish this.

## SQL Server Change Data Capture

This one is the easiest to setup and should be your first stop. When you enable Change Data Capture in SQL server, it will monitor the transaction log and record changes to change tables that mirror the structure of the original table. This can also be extended to an ETL process to update your data warehouse.

To see how this works, refer to [About Change Data Capture (SQL Server)](http://technet.microsoft.com/en-us/library/cc645937.aspx) on MSDN.

## Triggers

Before SQL Server 2008, change data capture did not exist. It was common to find a database that had triggers setup to capture changes to a table and update an audit table. Since you are setting up the triggers, it’s very easy to either copy the data to a table that mirrors the structure of the original table, or to copy the data into a more general audit log based table.

Jon Galloway has a good article, [Adding simple trigger-based auditing to your SQL Server database](http://weblogs.asp.net/jgalloway/archive/2008/01/27/adding-simple-trigger-based-auditing-to-your-sql-server-database.aspx), on how to set this up.

## Entity Framework

It’s also possible to have Entity Framework itself handle the audit logging. This scenario will typically come into play if your company has policies in place around SQL Server Data Capture that do not meet your applications needs or triggers are viewed as a bad practice and strongly discouraged in your organization. The downside of this approach is that auditing only works on changes flowing into your system through the code. Any SQL scripts run directly against the database will miss it. You’ll probably want an API, which is a good thing, because then everything will flow through your business rules.

To demonstrate how to do this in Entity Framework we are going to add four columns to each table for the simplest form of audit tracking that became popular with Ruby on Rails, and we are going to create an audit table like Jon Galloway does. You can choose to implement either or both of these methods in your application.

#### Setup Windows Authentication

First, we are going to change the authentication in the web project to use Windows Authentication, which is a popular option inside companies.

Replace this in the web.config:

<authentication mode="Forms">

  <forms loginUrl="~/Account/Login" timeout="2880" />

</authentication>

With this:

<authentication mode="Windows" />

<authorization>

  <deny users="?"/>

</authorization>

And this section to system.webServer:

<security>

  <authentication>

    <windowsAuthentication enabled="true"/>

    <anonymousAuthentication enabled="false"/>

  </authentication>

</security>

And add these keys to appSettings:

<add key="autoFormsAuthentication" value="false" />

<add key="enableSimpleMembership" value="false"/>

#### Basic Table Audit Tracking

Next, add the following four columns to each of the SQL files for tables in the database project:

[CreatedBy] NVARCHAR(50) NOT NULL,

[CreatedOn] DATETIME NOT NULL ,

[UpdatedBy] NVARCHAR(50) NULL,

[UpdatedOn] DATETIME NULL,

Update the TestData.sql file to include the CreatedBy and CreatedOn on each insert with the values ‘UnitTest’ and GETDATE():

INSERT INTO dbo.Person (LastName, FirstName, EnrollmentDate, Discriminator, CreatedBy, CreatedOn)

VALUES ('Alexander', 'Carson', '9/1/2010', 'Student', 'UnitTest', GETDATE());

DECLARE @Person\_Alexander INT = @@IDENTITY;

If you run the database project now to deploy the changes, you will get this error:

The schema update is terminating because data loss might occur.

Since we are re-populating a local database with each build, its okay to check the “Always re-create database” option under Database Properties > Debug.

Next we’ll update the models in Contoso University. Create an AuditBase class that has these four properties.

using System;

namespace ContosoUniversity.Models

{

    public abstract class AuditBase

    {

        public string CreatedBy { get; set; }

        public DateTime CreatedOn { get; set; }

        public string UpdatedBy { get; set; }

        public DateTime UpdatedOn { get; set; }

    }

}

And update each of the models to inherit from this base class. You can skip Instructor and Student because they already inherit from Person. Just update Person.

public class Course : AuditBase

Finally, we will add a SaveChanges method to the context class. Note the try wrapper is to protect against table that might not have or need the four audit columns.

public override int SaveChanges()

{

    var currentTime = DateTime.Now;

    var user = HttpContext.Current != null ? HttpContext.Current.User : Thread.CurrentPrincipal;

    foreach (var entity in ChangeTracker.Entries().Where(e => e.State == EntityState.Added   
 || e.State == EntityState.Deleted || e.State == EntityState.Modified))

    {

        try

        {

            dynamic dv = entity.Entity;

            if (entity.State == EntityState.Added)

            {

                dv.CreatedBy = user.Identity.Name;

                dv.CreatedOn = currentTime;

            }

            if (entity.State == EntityState.Modified)

            {

                dv.UpdatedBy = user.Identity.Name;

                dv.UpdatedOn = currentTime;

            }

        }

        catch { }

    }

    return base.SaveChanges();

}

There is an issue with the above code we need to resolve and a couple of ways we can do it. The create works just fine, but the update will be missing the create values. We could put the create and update values as hidden fields in the form or we can look up the original values before saving the object. We are going with the second option because next we will create an audit history of changes. This will put the object in a fully attached state. Every update method in the repositories should now look like this:

public void UpdateStudent(Student student)

{

    Student current = context.Students.Find(student.PersonID);

    context.Entry(current).State = EntityState.Modified;

    context.Entry(current).CurrentValues.SetValues(student);

}

The Generic repository needs to be updated to a similar pattern:

public virtual void Update(TEntity currentEntity, TEntity modifiedEntity)

{

    context.Entry(currentEntity).State = EntityState.Modified;

    context.Entry(currentEntity).CurrentValues.SetValues(modifiedEntity);

}

And also the Department controller:

if (ModelState.IsValid)

{

    Department current = db.Departments.Find(department.DepartmentID);

    db.Entry(current).State = EntityState.Modified;

    db.Entry(current).CurrentValues.SetValues(department);

    db.SaveChanges();

    return RedirectToAction("Index");

}

If you want, you can also clean up the forms authentication built into Contoso by removing the Account controller, login section from shared layout view, the account views, the login partial view and the default connection string in the web.config.

Re-generate the StudentContext.Views class and run the web site. Add a new student and then make sure the created columns are populated in the Person table. Then edit a student and make sure the updated columns are populated. Now is also a good time to re-run all the unit tests.

Next we can move on to create an actual audit history table.

#### Advanced Table Audit Tracking

Now we can add an audit table that tracks all changes to models in Entity Framework.

Add an Audit table to the Tables folder in the Database project:

CREATE TABLE [dbo].[Audit]

(

[AuditId] INT IDENTITY (1, 1)NOT NULL PRIMARY KEY,

    [TableId] INT NOT NULL,

    [User] NVARCHAR(50) NOT NULL,

    [TableName] VARCHAR(50) NOT NULL,

    [Action] VARCHAR(20) NOT NULL,

    [CreatedOn] DATETIME NOT NULL,

    [Before] XML NULL,

    [After] XML NOT NULL

)

Update the Reset script to reseed the identity for testing:

DELETE FROM dbo.Audit; DBCC CHECKIDENT('dbo.Audit', RESEED, 0);

Create an Audit class in the Models folder of the web project:

using System;

namespace ContosoUniversity.Models

{

   public class Audit

   {

       public int AuditId { get; set; }

       public int TableId { get; set; }

       public string User { get; set; }

       public string TableName { get; set; }

       public string Action { get; set; }

       public DateTime CreatedOn { get; set; }

       public string Before { get; set; }

       public string After { get; set; }

   }

}

Now let’s update the SchoolContext class:

using ContosoUniversity.Models;

using System;

using System.Collections.Generic;

using System.Data;

using System.Data.Entity;

using System.Data.Entity.Infrastructure;

using System.Data.Entity.ModelConfiguration.Conventions;

using System.Data.Objects;

using System.Linq;

using System.Text;

using System.Threading;

using System.Web;

using System.Xml;

namespace ContosoUniversity.DAL

{

   public class SchoolContext : DbContext

   {

       public DbSet<Course> Courses { get; set; }

       public DbSet<Department> Departments { get; set; }

       public DbSet<Enrollment> Enrollments { get; set; }

       public DbSet<Instructor> Instructors { get; set; }

       public DbSet<Student> Students { get; set; }

       public DbSet<OfficeAssignment> OfficeAssignments { get; set; }

       public DbSet<Person> People { get; set; }

       public DbSet<Audit> Audits { get; set; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)

        {

            modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();

            modelBuilder.Entity<Course>()

                .HasMany(c => c.Instructors).WithMany(i => i.Courses)

                .Map(t => t.MapLeftKey("CourseID")

                    .MapRightKey("PersonID")

                    .ToTable("CourseInstructor"));

        }

        public override int SaveChanges()

        {

            var currentTime = DateTime.Now;

            var user = HttpContext.Current != null ? HttpContext.Current.User : Thread.CurrentPrincipal;

            var changes = new List<Change>();

            foreach (var entity in ChangeTracker.Entries().Where(e => e.State == EntityState.Added || e.State == EntityState.Deleted || e.State == EntityState.Modified))

            {

                try

                {

                    dynamic dv = entity.Entity;

                    if (entity.State == EntityState.Added)

                    {

                        dv.CreatedBy = user.Identity.Name;

                        dv.CreatedOn = currentTime;

                    }

                    if (entity.State == EntityState.Modified)

                    {

                        dv.CreatedBy = entity.OriginalValues["CreatedBy"].ToString();

                        dv.CreatedOn = DateTime.Parse(entity.OriginalValues["CreatedOn"].ToString());

                        dv.UpdatedBy = user.Identity.Name;

                        dv.UpdatedOn = currentTime;

                    }

                }

                catch { }

                var change = new Change();

                change.State = ((IObjectContextAdapter)this).ObjectContext.ObjectStateManager.GetObjectStateEntry(entity.Entity);

                change.TableId = entity.State != EntityState.Added ? (int)change.State.EntityKey.EntityKeyValues[0].Value : 0;

                change.TableName = change.State.Entity.GetType().BaseType.Name == "Object" || change.State.Entity.GetType().BaseType.Name == "AuditBase" || change.State.Entity.GetType().BaseType.Name == "Person" ? change.State.Entity.GetType().Name : change.State.Entity.GetType().BaseType.Name;

                change.Action = entity.State.ToString();

                change.CurrentValues = entity.State != EntityState.Deleted ? entity.CurrentValues.Clone() : null;

                change.OriginalValues = entity.State != EntityState.Added ? entity.OriginalValues.Clone() : null;

                changes.Add(change);

            }

            var result = base.SaveChanges();

            foreach (var change in changes)

            {

                if (change.Action == "Added")

                    change.TableId = (int)change.State.EntityKey.EntityKeyValues[0].Value;

                var originalValues = new StringBuilder();

                var currentValues = new StringBuilder();

                var writer = XmlWriter.Create(originalValues);

                writer.WriteStartElement("values");

                if (change.OriginalValues != null)

                    foreach (var columnName in change.OriginalValues.PropertyNames)

                        writer.WriteElementString(columnName, change.OriginalValues[columnName] == null ? "" : change.OriginalValues[columnName].ToString());

                writer.WriteEndElement();

                writer.Flush();

                writer = XmlWriter.Create(currentValues);

                writer.WriteStartElement("values");

                if (change.CurrentValues != null)

                    foreach (var columnName in change.CurrentValues.PropertyNames)

                        writer.WriteElementString(columnName, change.CurrentValues[columnName] == null ? "" : change.CurrentValues[columnName].ToString());

                writer.WriteEndElement();

                writer.Flush();

                var audit = new Audit()

                {

                    TableId = change.TableId,

                    User = user.Identity.Name,

                    TableName = change.TableName,

                    Action = change.Action,

                    CreatedOn = currentTime,

                    Before = originalValues.ToString(),

                    After = currentValues.ToString()

                };

                Audits.Add(audit);

            }

            base.SaveChanges();

            return result;

        }

       private class Change

       {

           public int TableId { get; set; }

           public string TableName { get; set; }

           public string Action { get; set; }

           public DbPropertyValues CurrentValues { get; set; }

           public DbPropertyValues OriginalValues { get; set; }

           public ObjectStateEntry State { get; set; }

       }

   }

}

We have added the Audit class to context, but the real changes are all in the SaveChanges method.

1. We are pulling the changes entirely out of the context, which requires that we first convert the simpler DbContext to ObjectContext.
2. The TableId will not be generated for an Add until the database saves it, so that is 0 before the save, and populated after. This is one of the big reasons we are putting everything in a change list and then going back. You need to do this for any data that is generated by the database.
3. For the TableName we are protecting against grabbing AuditBase and Person instead of the table name.
4. Action is straight forward.
5. Original and Current values are written as XML. This could also be replaced with a related table of audit values instead.

Finally, create an AuditTest class in the test project for some basic testing, this is by no means complete:

using System;

using System.Linq;

using System.Web.Mvc;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using ContosoUniversity.Controllers;

using ContosoUniversity.DAL;

using System.Collections.Generic;

using ContosoUniversity.Models;

namespace ContosoUniversity.UnitTests

{

    [TestClass]

    public class AuditTest

    {

        [TestInitialize]

        public void Initiaize()

        {

            Helper.DataInitialize();

        }

        [TestMethod]

        public void CreateStudent()

        {

            var context = new SchoolContext();

            var controller = new StudentController();

            var expectedTableId = 14;

            var expectedTableName = "Student";

            var expectedAction = "Added";

            var student = new Student() { LastName = "Unit", FirstMidName = "Test", EnrollmentDate = new DateTime(2013,8,1) };

            controller.Create(student);

            Assert.AreEqual(expectedTableId, context.Audits.First().TableId);

            Assert.AreEqual(expectedTableName, context.Audits.First().TableName);

            Assert.AreEqual(expectedAction, context.Audits.First().Action);

        }

        [TestMethod]

        public void EditStudent()

        {

            var context = new SchoolContext();

            var controller = new StudentController();

            var expectedTableId = 1;

            var expectedTableName = "Student";

            var expectedAction = "Modified";

            var student = new Student() { PersonID = expectedTableId, LastName = "Unit Test", FirstMidName = "Test", EnrollmentDate = new DateTime(2013, 8, 1) };

            controller.Edit(student);

            Assert.AreEqual(expectedTableId, context.Audits.First().TableId);

            Assert.AreEqual(expectedTableName, context.Audits.First().TableName);

            Assert.AreEqual(expectedAction, context.Audits.First().Action);

        }

        [TestMethod]

        public void DeleteStudent()

        {

            var context = new SchoolContext();

            var controller = new StudentController();

            var expectedTableId = 1;

            var expectedTableName = "Student";

            var expectedAction = "Deleted";

            controller.Delete(expectedTableId);

            Assert.AreEqual(expectedTableId, context.Audits.First().TableId);

            Assert.AreEqual(expectedTableName, context.Audits.First().TableName);

            Assert.AreEqual(expectedAction, context.Audits.First().Action);

        }

    }

}

Re-generate the context views and run the web site again. Add a student, edit a student and delete a student. You should see three new records in the Audit table tracking each change. Re-run all your unit tests and make sure everything is still working.

You now have basic and detailed audit tracking built into Contoso University. If you want as an exercise, you can now extend the user interface to show this audit information.

## Clean Up

In all cases list above, it’s important that you don’t keep too much history in your live database. Audit tables can grow to be very large, very quickly. Make sure you define how much history you really need and move the rest into a historical database that can be tuned from both a software and hardware perspective to hold a large amount of data that is infrequently read. Or if you only need to keep a fixed amount (30 days, 1 year, etc) and that amount of data is acceptable, purge the old records.

# Part 5 – Custom Security

Windows authentication works well inside a corporate firewall, but the application team does not always have easy access to update domain account roles. Often you will find yourself wanting to maintain permissions inside the database. In the case of the Contoso University example we only have two different roles, student and instructor, stored in the database. So we’ll tap into that to make custom roles.

In .NET 4.5, Microsoft extended the idea of the custom principal and identity interfaces and implemented claims based authentication. We’ll use those and show how that integrates with anti-forgery tokens in MVC.

## The Database & Model

First, we need to extend the database to include windows user names and email addresses.

Add two columns to the Person table before the created and updated columns:

[UserName] NVARCHAR(256) NULL,

[Email] NVARCHAR(256) NOT NULL,

Update the TestData.sql to give everyone an email address. You can also give a teacher record your windows username for testing, but we’ll add a test controller to do the same thing. Example:

INSERT INTO dbo.Person (LastName, FirstName, EnrollmentDate, Discriminator, Email, CreatedBy, CreatedOn)   
VALUES ('Alexander', 'Carson', '9/1/2010', 'Student', 'alexander@contoso.edu', 'UnitTest', GETDATE());   
DECLARE @Person\_Alexander INT = @@IDENTITY;

Update the Person class with the same information:

[StringLength(256)]  
public string UserName { get; set; }

[StringLength(256)]

public string Email { get; set; }

Make sure you re-generate the context views, or you’ll get an error when you run the site or tests.

## Security

Create a Security folder in the Contoso University project and add a new class called CustomClaimsPrincipal. This class will inherit from the ClaimsPrinicipal in the Security Claims namespace. To get this working we need to:

1. Find the person by their Windows user name.
2. Add claims for their user name, name and email address.
3. Add a claim for their role.

using ContosoUniversity.DAL;

using ContosoUniversity.Models;

using System.Linq;

using System.Security.Claims;

using System.Security.Principal;

namespace ContosoUniversity.Security

{

    public class CustomClaimsPrincipal : ClaimsPrincipal

    {

        public CustomClaimsPrincipal(string userName)

        {

            GenericIdentity gi = new GenericIdentity(userName, "Contoso Univerity Custom Authentication");

            ClaimsIdentity ci = new ClaimsIdentity(gi);

            SchoolContext context = new SchoolContext();

            var person = new GenericRepository<Person>(context).Get().Where(p => p.UserName == userName).FirstOrDefault();

            if (person != null)

            {

                ci.AddClaim(new Claim(ClaimTypes.Name, person.FullName));

                ci.AddClaim(new Claim(ClaimTypes.WindowsAccountName, person.UserName));

                ci.AddClaim(new Claim(ClaimTypes.NameIdentifier, person.Email));

                ci.AddClaim(new Claim(ClaimTypes.Email, person.Email));

            }

            var instructor = new GenericRepository<Instructor>(context).Get()  
 .Where(p => p.UserName == userName).FirstOrDefault();

            if (instructor != null)

                ci.AddClaim(new Claim(ClaimTypes.Role, "Instructor"));

            var student = new GenericRepository<Student>(context).Get()  
 .Where(p => p.UserName == userName).FirstOrDefault();

            if (student != null)

                ci.AddClaim(new Claim(ClaimTypes.Role, "Student"));

            this.AddIdentity(ci);

        }

    }

}

We already have Windows Authentication up and working on the web site, so all we need to do is integrate our new custom principal. Create a SecurityModule class in the same folder. This class will get the Windows user name and pass it into the constructor of our new custom claims class.

using System;

using System.Threading;

using System.Web;

namespace ContosoUniversity.Security

{

    public class SecurityModule : IHttpModule, IDisposable

    {

        public void Init(HttpApplication application)

        {

            application.PostAuthenticateRequest += new EventHandler(this.PostAuthenticateRequest);

        }

        public void Dispose() { }

        public void PostAuthenticateRequest(Object source, EventArgs e)

        {

            if (HttpContext.Current.User.Identity.IsAuthenticated)

            {

                string userName = Thread.CurrentPrincipal.Identity.Name;

                CustomClaimsPrincipal cp = new CustomClaimsPrincipal(userName);

                Thread.CurrentPrincipal = cp;

                HttpContext.Current.User = cp;

            }

        }

    }

}

In a real scenario, you might want to store the user information in session after the first call. The last step is to wire up the module in the web.config file. Add a modules section after the handlers section in system.webServer:

<modules>

  <add name="SecurityModule" type="ContosoUniversity.Security.SecurityModule, ContosoUniversity" />

</modules>

The last step to wire up the web site is to configure anti-forgery. If you look at the code for one of the views that has a form, you’ll notice they start with @Html.AntiForgeryToken(). When you implement claims based authentication, you’ll need to tell the site what field you are using as a unique claim identifier. The easiest one to use is the email address. The custom claim class has already wired this in with the NameIdentifier and Email claim types. Update the Application\_Start method in the Global.asax.cs:

AntiForgeryConfig.UniqueClaimTypeIdentifier = ClaimTypes.Email;

Without that line of code, you’ll get this fun error message:

A claim of type 'http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier' or 'http://schemas.microsoft.com/accesscontrolservice/2010/07/claims/identityprovider' was not present on the provided ClaimsIdentity. To enable anti-forgery token support with claims-based authentication, please verify that the configured claims provider is providing both of these claims on the ClaimsIdentity instances it generates. If the configured claims provider instead uses a different claim type as a unique identifier, it can be configured by setting the static property AntiForgeryConfig.UniqueClaimTypeIdentifier.

## Controllers and Views

Add the following attribute to the Create, Edit and Delete posts just under the ValidateAntiForgeryToken attribute:

[Authorize(Roles="Instructor")]

You can use this to test your changes. If you want you can update all the form posts to require either the instructor or student role. You will also need to add the email address to the Edit post:

if (TryUpdateModel(instructorToUpdate, "", new string[] { "LastName", "FirstMidName", "Email", "HireDate", "OfficeAssignment" }))

The student controller will also need the email address added to the form posts.

[Bind(Include = "LastName, FirstMidName, Email, EnrollmentDate")]Student student

Finally, you need to update all the view to pass in an email address since it is required. For example, the Instructor create view would have this added:

<div class="editor-label">

    @Html.LabelFor(model => model.Email)

</div>

<div class="editor-field">

    @Html.EditorFor(model => model.Email)

    @Html.ValidationMessageFor(model => model.Email)

</div>

Copying the FirstMidName field in each view should do the trick.

## Testing

To test the website, add a controller to the home page:

public void TestUser()

{

    var instructor = db.Instructors.First();

    instructor.UserName = User.Identity.Name;

    db.SaveChanges();

}

You can call this controller using the url: /Home/TestUser to assign your windows account to the first instructor.

Update the Helper class DataInitialize method in the test project to add a userLastName parameter set to an empty string:

public static void DataInitialize(string userLastName = "")

and include the following code immediately after cmd.ExecuteNonQuery() call:

if (!string.IsNullOrEmpty(userLastName))

{

    var updatePerson = new SqlCommand("UPDATE Person SET UserName = @UserName WHERE LastName = @LastName", cn);

    updatePerson.Parameters.AddWithValue("@UserName", Thread.CurrentPrincipal.Identity.Name);

    updatePerson.Parameters.AddWithValue("@LastName", userLastName);

    updatePerson.ExecuteNonQuery();

}

This will let you pass in the last name of a person and tie it to your windows account.

Now you can create a new Security test class:

using ContosoUniversity.Security;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using System.Security.Claims;

using System.Threading;

namespace ContosoUniversity.UnitTests

{

    [TestClass]

    public class SecurityTest

    {

        [TestInitialize]

        public void Initiaize()

        {

            Helper.DataInitialize("Abercrombie");

        }

        [TestMethod]

        public void CustomClaimsPrincipal()

        {

            var claim = new CustomClaimsPrincipal(Thread.CurrentPrincipal.Identity.Name);

            Assert.IsTrue(claim.HasClaim(ClaimTypes.Email, "abercrombie@contoso.edu"));

            Assert.IsTrue(claim.IsInRole("Instructor"));

        }

    }

}

When you rerun all the tests, a lot of them are going to fail because email address is required. Update the other unit tests to add an email address.

# More

Claims based authentication.

<http://www.code-magazine.com/Article.aspx?quickid=1309101>

<http://visualstudiomagazine.com/articles/2013/08/01/leveraging-claims-based-security-in-aspnet-45.aspx>

# Part 6 – Performance

We already looked at enhancing performance through the use of generated views and the Contoso University sample also discusses some performance optimization tricks such as turning off tracking. So in this section we are going to discuss performance tracking and testing.

## Tracking

Windows and SQL Server have a lot of different ways you can track performance. And there are a lot of third party tools to help you track and monitor performance. The two most common methods are performance counters and profiling. I’m going to show you an easy way to add an Entity Framework based simple performance logger.

First we need to create the table. You could write to a text file or the event log, but we’re going to use a table because it’s the easiest way for you to later create reports.

CREATE TABLE [dbo].[PerformanceLog]

(

[PerformanceLogId] INT IDENTITY (1, 1) NOT NULL PRIMARY KEY,

    [Action] NVARCHAR(20) NOT NULL,

    [Model] NVARCHAR(50) NOT NULL,

    [Query] NVARCHAR(MAX) NOT NULL,

    [Date] DATETIME NOT NULL,

    [Execution] BIGINT NOT NULL,

    [User] NVARCHAR(50) NULL,

    [IPAddress] NVARCHAR(15) NULL,

    [RequestUrl] NVARCHAR(MAX) NULL

)

And update the Reset SQL script:

DELETE FROM dbo.PerformanceLog; DBCC CHECKIDENT('dbo.PerformanceLog', RESEED, 0);

Now add the model class:

using System;

namespace ContosoUniversity.Models

{

   public class PerformanceLog

   {

       public int PerformanceLogId { get; set; }

       public string Action { get; set; }

       public string Model { get; set; }

       public string Query { get; set; }

       public DateTime Date { get; set; }

       public long Execution { get; set; }

       public string User { get; set; }

       public string IPAddress { get; set; }

       public string RequestUrl { get; set; }

   }

}

Most of the work is going to be done in a Timer class added to the DAL namespace:

using ContosoUniversity.Models;

using System;

using System.Diagnostics;

using System.Threading;

using System.Web;

namespace ContosoUniversity.DAL

{

    public class Timer : IDisposable

    {

        private SchoolContext \_context;

        readonly Stopwatch \_stopWatch = new Stopwatch();

        private string \_action;

        private string \_model;

        private string \_query;

        public Timer(SchoolContext context, string action, string model, string query)

        {

            \_context = context;

            \_action = action;

            \_model = model;

            \_query = query;

            \_stopWatch.Reset();

            \_stopWatch.Start();

        }

        public void Dispose()

        {

            \_stopWatch.Stop();

            var perfLog = new PerformanceLog()

            {

                Action = \_action,

                Model = \_model,

                Query = \_query,

                Date = DateTime.Now,

                Execution = \_stopWatch.ElapsedMilliseconds,

                User = HttpContext.Current != null ?   
 HttpContext.Current.User.Identity.Name : Thread.CurrentPrincipal.Identity.Name,

                IPAddress = HttpContext.Current != null ? HttpContext.Current.Request["REMOTE\_ADDR"] : null,

                RequestUrl = HttpContext.Current != null ? HttpContext.Current.Request.Url.ToString() : null

            };

            \_context.PerformanceLog.Add(perfLog);

            \_context.SaveSimpleChange();

        }

    }

}

There are a few things to note here. First, we are using the StopWatch class for a high precision timer. Second, the timer is completely configured inside the construction. Third, the logging happens in the disposal. The reason we doing the last two this way, is so that we can simply wrap our EF execution in a using statement.

Now let’s update the SchoolContext by adding a new SaveChange method referenced in the Timer class that skips all the audit logging.

public int SaveSimpleChange()

{

    return base.SaveChanges();

}

And add the PerformanceLog class to the list of DbSets:

public DbSet<PerformanceLog> PerformanceLog { get; set; }

Finally we can start tracking performance. First, we want to track the time it takes to save any changes to the database. This can easily be caught in a single spot in SchoolContext.SaveChanges. There are two base.SaveChanges calls in this method, we want to replace the first one that updates the database, not the second one that records audit logging.

int result = 0;

using (var timer = new Timer(this, "SaveChanges", "Multiple", perfEntities))

{

    result = base.SaveChanges();

}

Next we want to record the time it takes to make reads. We are going to update the Generic Repository for this. The following is a list of Timer configurations that should wrap calls in this class. To see the full class updated, you can download the source code.

GetWithRawSQL Timer(context, "GetWithRawSql", typeof(TEntity).Name, query)

Get Timer(context, "Get", typeof(TEntity).Name, query.ToString())

GetById Timer(context, "GetByID", typeof(TEntity).Name, id.ToString())

You would also need to add similar statements to custom repositories and the controllers to catch everything. The Contoso Univerisity example was setup to show a variety of ways to use Entity Framework. In a well-structured application, you would have layers that flow into fewer data access points. I’ll cover this in more detail on the section about Business Rules.

But this simple Timer class example shows you an easy way to record performance stats.

### Extending the Example

* It’s a good idea to setup an application setting in the web.config or another way to disable performance tracking when you need to make the application perform as fast as possible.
* Make sure your database log table is designed for high performance writing, not reading.
* Archive or truncate your performance log table periodically, don’t let it grow forever.
* You could use a similar Timer class or extend the existing one to track the time it takes for the entire controller to execute.
* Create reports based on the performance log table that tracks the worst performing requests, database calls per page request, and even how well the system scales based on how many requests are happening at once.

## Performance Testing

Web Performance and Load Testing in Visual Studio are great tools, but they only come in the more expensive editions. Even if you are using these, it’s still a good idea to add some basic performance testing to your unit tests. The easiest way to do this to follow a similar approach with tests. Add the following to the Unit Test Helper class:

readonly static Stopwatch \_stopWatch = new Stopwatch();

public const long TimeStandard = 1000;

public static void StartTimer()

{

    \_stopWatch.Reset();

    \_stopWatch.Start();

}

public static long StopTimer()

{

    \_stopWatch.Stop();

    return \_stopWatch.ElapsedMilliseconds;

}

This code defines a single time value to test against of 1 second. You could add more variations such as a TimeWeb for measuring controllers.

To use the code just wrap your Act code in a timer and then add an Assertion. The following code records a slow response as warning instead of failing the test.

Helper.StartTimer();

controller.UpdateCourseCredits(multiplier);

var time = Helper.StopTimer();

….

if (time > Helper.TimeStandard)

Assert.Inconclusive("Time = " + time + ", Expected <= " + Helper.TimeStandard);

This is some pretty basic performance testing, but it goes a long way toward catching problem queries. It also helps to point this type of test against a database loaded with data. Almost every query runs fast against a small set of data. You’ll want to consider creating a test plan of just performance tests to run against this larger data set.

# Part 7 – Business Rules

Contoso University was designed to demonstrate the power and flexibility of Entity Framework and how you can work with it in MVC. It’s not designed as a well-structured, well layered business application. The Context class is accessed by GenericRepository, Student Repository which does not inherit from the GenericRepository, and several Controller classes directly. Sometimes the UnitOfWork class comes into play and sometimes it doesn’t.



In a real world scenario, I would structure this application so that everything flows through a Façade layer which would look a lot like the existing Controller classes with no MVD specific code. The Façade layer would access the the GenericRepository or a specific Repository that inherits from GenericRepository. You could leave the UnitOfWork or eliminate it and let the Façade layer handle it.



What this buys us right away are a few things.

1. The performance tracking could now be centralized on just the repositories.
2. Simple business rules, such as complex types, can still live on the models, but workflows or more complex rules would be contained within the Façade layer.
3. You could add more interfaces such as Web API without duplicating logic stored in the MVC controllers.

This is a very basic architecture, and you might consider further implementing more patterns such as Dependency Injection and IoC containers. But this provides a solid foundation to move forward.

# Part 8 – Entity Framework 6

As I write this series, Entity Framework 6 is currently in Release Candidate mode. There are at least 3 features that have an impact on the sample code we have been building and how it was designed for Entity Framework 5.

## Async Query and Save

The SaveChanges base method calls inside SchoolContext.SaveChanges can be made asynchronous. There is no need to wait on the first SaveChanges to complete before creating the Audit records. In both the existing example and after making this change, there is a chance an error could get this tables out of sync. Depending on how important it is to your app that these be 100% accurate, you should combine the two writes into a single transaction.

The Timer class should definitely do an asynchronous save so the rest of the workflow can move on while its updating the log. But this is even easier to deal with using the new SQL Logging feature.

## Interception / SQL Logging

The Timer class can be updated or eliminated. Instead you can centralize performance tracking and logging in Entity Framework 6 using Database.Log on the context. This captures all the SQL commands including execution time. You will still need to parse out the SQL from the execution times

## Testablity

The tests created for the example were only meant to keep us from breaking things while updating Contoso University. They are not great real world examples. One of the issues I have found when dealing with Entity Framework is unit testing. You can test your application using Entity Framework by creating a sample database with a minimal set of data. But these tests take longer to run because you are loading up your test data continuously. Now you will be able to create a mock implementation for DbContext and avoid reading and writing to the database.

# What’s Left

There are a few more things I wanted to touch on that are beyond the scope of Entity Framework but are likely going to come up when dealing with an enterprise application.

## Health Monitoring and Error Handling

One thing I didn’t discuss was health monitoring or what to do when things go bad. You could easily follow the same approach to log errors as the way we implemented performance logging. However, in this case you would save a lot of time by using a third party logging library like NLog. Through some simple configuration and trapping unhandled exceptions you could write errors to multiple sources like the event log, database and files as well as send out email alerts.

## API and Other Interfaces

When your application is structured with an organized layered approach as addressed in the Business Rules section, it’s easy to add a new interface such as through Web API. On a project I am working on currently, we structured the repositories to match the oData specification. Our API exposes these calls and on the client side we use Microsoft Excel to create an ad-hoc interface that does not bypass the business rules to go directly to the database. This approach can also extend to integrating with other internal system, adding a desktop interface or implementing a mobile application.