UNIT TESTING ENTITY FRAMEWORK DBCONTEXT **(VIA EFFORT.EF6)**

Contents

[SUMMARY AND JUSTIFICATION: 1](#_Toc522024689)

[USAGE EXAMPLE - SIMPLE 2](#_Toc522024690)

[TIPS: 4](#_Toc522024691)

[RESOURCES: 5](#_Toc522024692)

[REFERENCES: 5](#_Toc522024693)

[ADVANCED SCENARIOS 5](#_Toc522024694)

[CREATING LOCAL DATA 5](#_Toc522024695)

[CREATING DATA WITH DATA LOADER: 6](#_Toc522024696)

[UTILIZING CSV FILES IN UNIT TESTS 8](#_Toc522024697)

# SUMMARY AND JUSTIFICATION:

Effort: Entity Framework Fake ObjectContext Realization Tool

The Effort library works with DbContext objects by emulating the relational database server with a completely .NET based lightweight in-memory database. All the data operations are redirected to it, so the tests can run completely in-process.

Anyone who's tried to mock a DbContext can attest to the difficulty -- especially when approaches like the Repository Pattern have been incorporated into the application. Using Moq to test via an interface is fairly simple. But keep in mind that under the hood EF has mechanics for many things, like: generating ids, entity state tracking, virtual collection loading etc. Those cannot be quickly mocked, and from the unit test perspective it is worthless to spend a lot of time on mocking rather than on testing.

This is where the Effort library comes in. It emulates the relational database server with a .NET-based lightweight and in-memory database. All the data operations are redirected to it so that the tests can run completely in-process. This isolation is especially helpful when running tests on a continuous build server. <https://blog.goyello.com/2016/07/14/save-time-mocking-use-your-real-entity-framework-dbcontext-in-unit-tests/>

Note that using Effort in all unit test scenarios is not needed. It was specifically created with Entity Framework DbContext test issues. Other unit test frameworks can take care of many other scenarios.

# USAGE EXAMPLE - SIMPLE

The following simplified example utilizes a WebAPI application written in .NET 4.6.1 using a repository pattern with code-first Entity Framework.

1. Using Visual Studio template for “ASP..NET Web Application,” create WebApi app and include a model class and repository class with interface.

|  |
| --- |
| public class Game  {  public Game()  {  }  public Game(string title)  {  Title = title;  }  public int Id { get; set; }  public string Title { get; set; }  } |
| public class GameRepository : IGameRepository  {  private readonly MediaContext \_context;  public GameRepository(MediaContext context)  {  \_context = context;  }  public Game Get(int id)  {  return \_context.Games.SingleOrDefault(x => x.Id == id);  }  public void Insert(Game game)  {  \_context.Games.Add(game);  \_context.SaveChanges();  }  } |
| public interface IGameRepository  {  Game Get(int id);  void Insert(Game game);  } |

1. Create a DbContext implementation:

|  |
| --- |
| public class MediaContext : DbContext  {  public MediaContext()  {  }  public DbSet<Game> Games { get; set; }  } |
|  |

1. Components Installed:
   1. In main project:
      1. Install-Package EntityFramework -Version 6.1.3
   2. In test project:
      1. Install-Package Effort.EF6 -Version 1.3.5 (Also adds reference for NMemory)
2. Execute 'Enable-Migrations.' Note that we are not going to actually use the migration for database purposes but instead will utilize the functionality it creates. In Package Manager Console, run the command:

|  |
| --- |
| Enable-Migrations |

Reference: <https://msdn.microsoft.com/en-us/library/jj591621(v=vs.113).aspx>

1. Create an additional DBContext constructor to include 'DbConnection', true params. This will allow the 'connection' to be disposed. ( public MediaContext(DbConnection connection) : base(connection, true)) This constructor will allow us to pass in the connection created by Effort.

|  |
| --- |
| public class MediaContext : DbContext  {  public MediaContext()  {  }  //Constructor for unit testing with Effort  public MediaContext(DbConnection connection) : base(connection, true)  {  }  public DbSet<Game> Games { get; set; }  } |
|  |

1. Create and run tests. Note: If 'migrations' error occurs, then exclude the Configuration.cs Migrations file from main project. This does break the migration configuration but this is ok because we're not using it in this scenario. Reference: <http://www.vannevel.net/2015/02/26/11/>

|  |
| --- |
| [TestClass()]  public class GameRepositoryTests  {  private MediaContext \_context;  private IGameRepository \_repository;  private string \_title;  /\*  \* Note that the connection object is created in the Initialize() method  \* and passed to the GameContext and subsequently the GameRepository.  \* In essence this means that, since the connection is local and the  \* context and repository are overwritten after each test,  \* the connection and database will be disposed after each test.  \* This results in entirely separated database instances for each test  \*/  [TestInitialize]  public void Initialize()  {  /\*  \*NOTE: Create the db connection with the .CreateTransient option.  \*This will allow tin in-memory db instance to live during the object  \*lifecycle and be disposed of  \*/  var connection = DbConnectionFactory.CreateTransient();  \_context = new GameRepository(\_context);  \_title = "Halo Combat Evolved";  }  [TestMethod()]  public void GetGame\_WithoutExistingId()  {  // Arrange  const int nonExistingId = 117;  // Act  var book = \_repository.Get(nonExistingId);  // Assert  Assert.IsNull(book);  }  [TestMethod()]  public void AddGame()  {  // Arrange  var game = new Game(\_title);  // Act  \_repository.Insert(game);  // Assert  Assert.IsTrue(\_context.Games.Count() == 1);  Assert.AreEqual(\_context.Games.First().Title, \_title);  }  [TestMethod()]  public void GetGameById()  {  // Arrange  var game = new Game("Halo Combat Evolved");  \_repository.Insert(game);  // Act  var result = \_repository.Get(game.Id);  // Assert  Assert.AreEqual(result.Title, \_title);  Assert.AreEqual(result.Id, game.Id);  }  } |

### TIPS:

1. When using code first, make use of the DbConnection and DbConnectionFactory classes and NOT the EntityConneciton and EntityConnectionFactory classes.
2. When running multiple tests with Effort and the following errors occur: System.InvalidOperationException: The Entity Framework was already using a DbConfiguration instance before an attempt was made to add a 'Loaded' event handler. 'Loaded' event handlers can only be added as part of application start up before the Entity Framework is used.

Solution: Call the Effort.Provider.EffortProviderConfiguration.RegisterProvider() method at entry point of the problematic test

1. Limitations: Effort does not work with pure SQL commands; ie, \*.ExecuteCommand.
2. Likewise, using direct LINQ to SQL can also cause caching issues if the DbConnectionFactory is not set up properly.

### RESOURCES:

GitHub:

<https://github.com/zzzprojects/EntityFramework-Effort>

Nuget:

<https://www.nuget.org/packages/Effort/>

Project Site:

<http://entityframework-effort.net/?z=codeplex>

### REFERENCES:

<https://blog.goyello.com/2016/07/14/save-time-mocking-use-your-real-entity-framework-dbcontext-in-unit-tests/>

<https://tflamichblog.wordpress.com/category/effort/>

<http://www.vannevel.net/2015/02/26/11/>

# ADVANCED SCENARIOS

### CREATING LOCAL DATA

When you need to test data from a disconnected build server and you need data, there are at least a few options. One option is that data can be created by hand. As another option, data can be mocked with a framework such as Autofixture. With the Effort library, data loader tools are available to specifically create structures and their data that are similar to the real-time database.

Offhand, this can seem cumbersome; however, the data loader is a useful component in the Effort library specifically designed to help setting up the initial state of a fake database. Adding records to the tables by using the Entity Framework API can be inflexible and the written code might become hard to maintain. Furthermore these type of insert operations flow through the entire EF and Effort pipeline. This might have great performance impact. Data loaders solves these problem by allowing to insert data from any custom source during the initialization with extra small overload. REFERENCE: <https://tflamichblog.wordpress.com/2013/01/22/data-loaders-in-effort/>

Currently, there are three types of data loaders available:

1. **EntityDataLoader**: able to fetch data from an existing database by utilizing an existing Entity Framework compatible ADO.NET provider. It is initialized with an entity connection string.

|  |
| --- |
| EXAMPLE:  var dataLoader = new EntityDataLoader("name=MyEntities");  var connection = DbConnectionFactory.CreateTransient(dataLoader); |

1. **CsvDataLoader**: purpose is to read data records from CSV files. It is initialized with a path that points to a folder containing the CSV files. Each file represents the content of a database table.

|  |
| --- |
| EXAMPLE:  var dataLoader = new CsvDataLoader(@"C:\path\to\files"); |

The following example utilizes the CsvDataLoader. The CSV file format descriptors can be found here: <https://tflamichblog.wordpress.com/2013/01/22/the-csv-file-format-in-effort/>

1. **CacheDataLoader**: designed to speed up the initialization process by wrapping any kind of data loader with a cache layer.

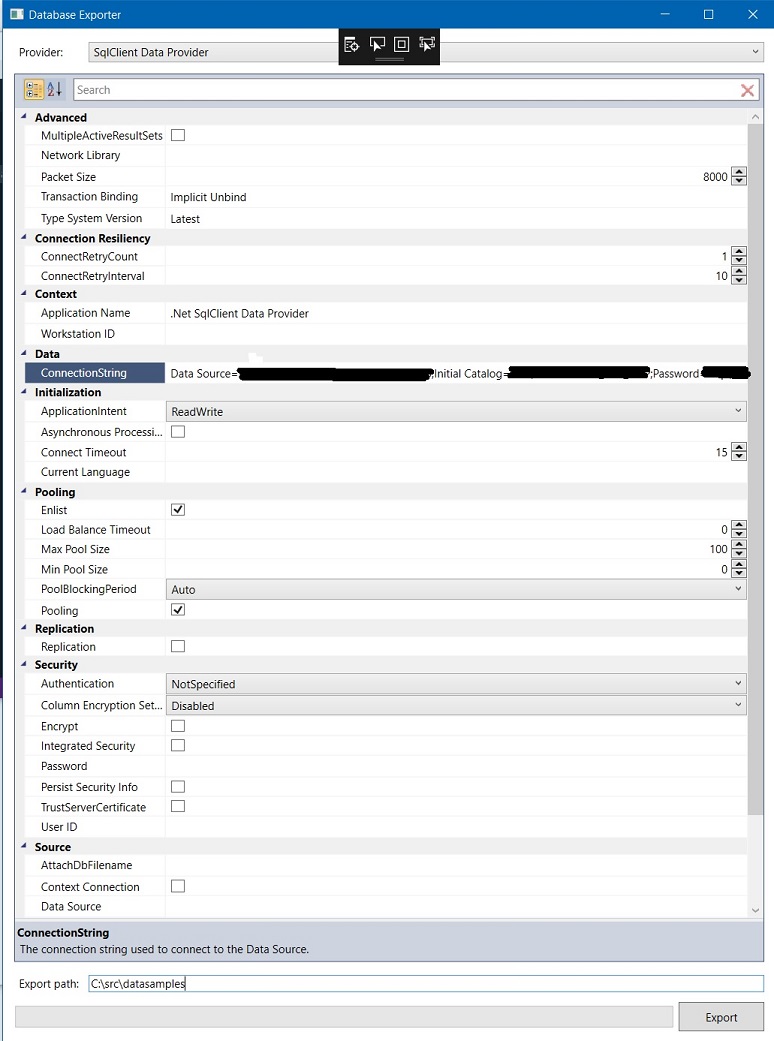
|  |
| --- |
| EXAMPLE:  var wrappedDataLoader = new CsvDataLoader(@"C:\path\to\files");  var dataLoader = new CachingDataLoader(wrappedDataLoader, false); |

Code and data loader tools can be found here:

<https://github.com/zzzprojects/EntityFramework-Effort>

### CREATING CSV DATA WITH DATA LOADER:

1. Create data files per the above-referenced CSV format. Each file will represent a table in your dataset and should be named accordingly.
2. The tool itself will look like the following:



1. The tool will export all tables and data in the catalog that you choose. Do NOT export production data.
2. Once the files are exported, it is recommended to:
   1. Remove tables that are not used in your DbContext
3. Reduce and/or alter data to a feasible size and content.

### UTILIZING CSV FILES IN UNIT TESTS

1. First, on the initialization of your DbContext, provide a path to your data files.

|  |
| --- |
| var path = @"C:\repo\MyProject\EffortData";  DbConnection connection = DbConnectionFactory.CreateTransient(new CsvDataLoader(path));  DbAuthContext = new AuthorizationContext(connection); |

1. Second, create a DbContext constructor as follows:

|  |
| --- |
| 1. /// <summary> 2. /// Constructor used for unit tests utilizing EntityFramework.Effort 3. /// </summary> 4. /// <param name="connection"></param> 5. public MyProjectContext(DbConnection connection) : base(connection, true) 6. { 7. } |

ADVANCED APPLICATION (authorization.api)

In this scenario, the following setup exists:

* WebAPI - .NET 4.6
* Entity Framework – Code First
* A base controller with sub-controllers
* A base repository with sub-repositories

1. In the base API controller, the DbContext is initialized as follows:

|  |
| --- |
| public BaseApiController()  {  if (GlobalManager.DbContextDbSource)  {  //When running as live application then create DbContext with the actual database connection string  DbAuthContext = new AuthorizationContext(GlobalManager.Utils.GetConnectionString("AuthEntities"));  }  else  {  //When running from unit tests, then create DbContext utilizing the Effort DbConnectionFactory  var path = @"C:\repo\cdp.authorization.ci\cdp.authorization.api\EffortData";  DbConnection connection = DbConnectionFactory.CreateTransient(new CsvDataLoader(path));  DbAuthContext = new AuthorizationContext(connection);  //TODO: hard-code just for now then remove  DbAuthContext.FilteredCdpId = 3455;  DbAuthContext.CdpIdName = "APPTEST";  }  } |

1. Note that the path to the test data is initiated in the main project. This allows for the external data to be loaded at once and applied through the project. Typically, this action would occur in the test project.
2. There are multiple DbContext constructors:

|  |
| --- |
| /// <summary>  /// Constructor used for unit tests utilizing EntityFramework.Effort  /// </summary>  /// <param name="connection"></param>  public AuthorizationContext(DbConnection connection) : base(connection, true)  {  }  /// <summary>  /// Constructor used for live app  /// </summary>  /// <param name="ConnectionString"></param>  public AuthorizationContext(string ConnectionString)  : base(ConnectionString)  {    } |