EF: Data Contexts - DbContexts

When working in EF, you will come across the need to create a DbContext type.

MS has a base class that you inherit from, called: DbContext.

It is useful to implement either a two or three layer DbContext.

See this page for EF Logging implementation: EF: Logging

Two Layer DbContext

The two layers in this organizational pattern include:

- A base type with common elements.
- A top-level type with provider-specific details.

The base type inherits from DbContext, and bolts in dynamic entity registration and EF log management.

And, the top layer adds in the provider-specific logic for the backing store.
This would include logic for composing connection strings for SQL Server, Postgres, SQLite, etc.

Three Layer DbContext

The three layer pattern for DbContext adds a middle layer that separates out the domain-specific logic.

Here are the three layers:

- Base Layer - common elements and logic
- Domain Layer - A middle layer for domain registrations
- Provide Layer - A top-level type with provider-specific details

Base Layer DbContext

This layer inherits from DbContext.

It includes common logic for all data context types.

It includes properties and logic to manage EF logging.
This includes controlling the logging level and a delegate for directing log output.

Here's what the base layer data context would look like:

```
using System;
using System.Diagnostics.CodeAnalysis;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Logging;

namespace DbContextLibrary
{
    /// <summary>
    /// Wraps the base db context class to provide late-bound entity mapping and date setting on save.
    /// To, use, derive a context class from this base that includes your DbSet props. Then, derive a datastore-specific type from that for each storage type you have: PostGres, MSSQL, InMem, etc...
    /// NOTE: The OnModelCreating override, in this class, makes use of AssemblyHelper to prescreen and compose a list of assemblies where IEntityTypeConfiguration<> implementations can be found.
    /// NOTE: So, be sure, during early process startup, to set the AssemblyHelper_Base.AssemblyHelperRef with a valid AssemblyHelper instance.
    /// </summary>
    public class cDBDContext_Base : DbContext
    {
        #region Private Fields

        protected string _classname;

        #endregion



        #region Public Properties

        /// <summary>
        /// Indicates what database implementation this context maps to: MSSSQLSERVER, POSTGRES, SQLITE, etc...
        /// </summary>
        public string DatabaseType { get; protected set; } = "";

        /// <summary>
```

```
        /// Behavior property that affects whether or not, this class and derivatives will pull in all implementations of IEntityTypeConfiguration<>, from the process's referenced assemblies.
        /// If not aware, these are the custom mapping models that tell EF how to map entity properties to table columns, what value conversions to perform, etc.
        /// This property (behavior) is enabled by default.
        /// If you don't want this behavior to occur in your implementation, set this property to false in the constructor of the derived class.
        /// NOTE: Leaving this enabled doesn't cause trouble for stray mappings to be pulled into a database context, because only the mappings of a DbSet will be used.
        /// </summary>
        public bool Cfg_LoadAllMapConfigsfromAssemblies { get; set; } = true;

        /// <summary>
        /// Set this if you want EF to dump logs.
        /// </summary>
        static public bool Cfg_Enable_EFLogging { get; set; } = false;

        /// <summary>
        /// If you hav enabled EF logging, you can assign a callback, here, to redirect logs.
        /// Logs are dumped to console, if this is unset.
        /// </summary>
        static public Action<string>? Cfg_LogTarget { get; set; } = null;

        /// <summary>
        /// This defaults logging to informational, if logging is enabled.
        /// </summary>
        static public LogLevel Cfg_LogLevel { get; set; } = LogLevel.Information;

        #endregion



        #region ctor / dtor

        public cDBDContext_Base([NotNullAttribute] DbContextOptions options) : base(options)
        {
            var tt = this.GetType();
            this._classname = tt.Name;

            OGA.SharedKernel.Logging_Base.Logger_Ref?.Info(
                $"{_classname}:DataContext - started.");
```

```
            OGA.SharedKernel.Logging_Base.Logger_Ref?.Info(
                $"{_classname}:DataContext - completed.");
        }

        public cDBDContext_Base() : base()
        {
            var tt = this.GetType();
            this._classname = tt.Name;

            OGA.SharedKernel.Logging_Base.Logger_Ref?.Info(
                $"{_classname}:DataContext - started.");

            OGA.SharedKernel.Logging_Base.Logger_Ref?.Info(
                $"{_classname}:DataContext - completed.");
        }



        #endregion



        #region Private Methods

#if NET5_0
        // NOTE: The convention converter in this block is only available in EF6 and forward.
        // So, EF5 usage will require individual value converters for each DateTime property in the model builder logic of each entity.
#else
        // NOTE: The convention converter in this block is only available in EF6 and forward.
        // So, EF5 usage will require individual value converters for each DateTime property in the model builder logic of each entity.



        /// <summary>
        /// This was added to globally retrieve all stored DateTime properties with their UTC flag set.
        /// If your implementation of classes has a mix of UTC and local time properties, you will need to be
        /// more surgical, and use individual value converters instead of this override.
        /// If this is the case, override this method (in your derived class) to be blank and not call the base, and assign individual value converters in the appropriate model builder instances.
        /// See this usage wiki:
```

```
https://oga.atlassian.net/wiki/spaces/~311198967/pages/66322433/EF+Working+with+DateTime
        /// </summary>
        /// <param name="configurationBuilder"></param>
        protected override void ConfigureConventions(ModelConfigurationBuilder configurationBuilder)
        {
            configurationBuilder
                .Properties<DateTime>()
                .HaveConversion<DateTimeUTCConverter>();
        }

#endif
        /// <summary>
        /// This override retrieves all the IEntityTypeConfiguration<> implementations in process assemblies, and
makes them available as entity mappings.
        /// If you don't want this behavior, set Cfg_LoadAllMapConfigsfromAssemblies = false in your derived class'
constructor.
        /// You can
        /// </summary>
        /// <param name="modelBuilder"></param>
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            // Search for IEntityTypeConfiguration<> implementations if required...
            if(Cfg_LoadAllMapConfigsfromAssemblies)
            {
                // Iterate all assemblies, and search for types that implement IEntityTypeConfiguration, and register
each one.
                var asl = OGA.SharedKernel.Process.AssemblyHelper_Base.AssemblyHelperRef.Get_All_Assemblies();
                foreach (var fff in asl)
                {
                    modelBuilder.ApplyConfigurationsFromAssembly(fff);
                }
            }

            base.OnModelCreating(modelBuilder);
        }

        /// <summary>
        /// This override lets us convert the Options instance we got at construction,
        ///     and convert it to a DbContextOptions, which is required by the DbContext base.
        /// This override is called on first usage of the context, not at construction.
```

```
        /// </summary>
        /// <param name="options"></param>
        protected override void OnConfiguring(DbContextOptionsBuilder options)
        {
            // Dump logs if asked...
            if(Cfg_Enable_EFLogging)
            {
                options.EnableSensitiveDataLogging(); // shows parameter values
                options.LogTo(LogInfeed, Cfg_LogLevel);
            }
        }



        /// <summary>
        /// This method protects the dbcontext, by allowing it to always have a logging target method, to call.
        /// And, this method decides what that target points to, based on runtime-accessible config.
        /// We include this indirection, to allow the logging target to be changed at runtime, without causing problems in the db context.
        /// </summary>
        /// <param name="msg"></param>
        private void LogInfeed(string msg)
        {
            try
            {
                if(Cfg_LogTarget != null)
                    Cfg_LogTarget(msg);
                else
                    Console.WriteLine(msg);
            } catch (Exception) { }
        }

        #endregion
    }
}
```

Domain Layer DbContext

Provider Layer DbContext

Revision #3
Created 17 November 2025 23:08:44 by glwhite
Updated 19 February 2026 06:44:47 by glwhite