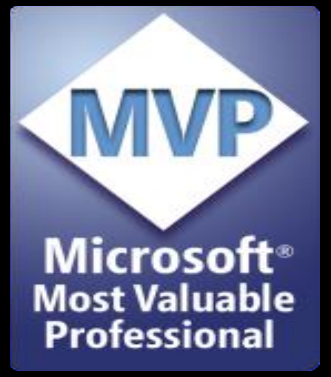
ENTITY FRAMEWORK 6.X – WHAT YOU NEED TO KNOW

Philip Japikse (@skimedic)
skimedic@outlook.com
www.skimedic.com/blog
Microsoft MVP, ASPInsider, MCSD, MCDBA, CSM, CSP
Principal Consultant/Architect, Strategic Data Systems



Phil.About()

- Principal Consultant/Architect, Strategic Data Systems
  - http://www.sds-consulting.com
- Microsoft MVP, ASPInsider, MCSD, MCDBA, CSM, CSP
- Founder, Agile Conferences, Inc.
- President, Cincinnati .NET User's Group
- Co-host, Hallway Conversations
  - www.hallwayconversations.com

Strategic Data Systems – What we do:

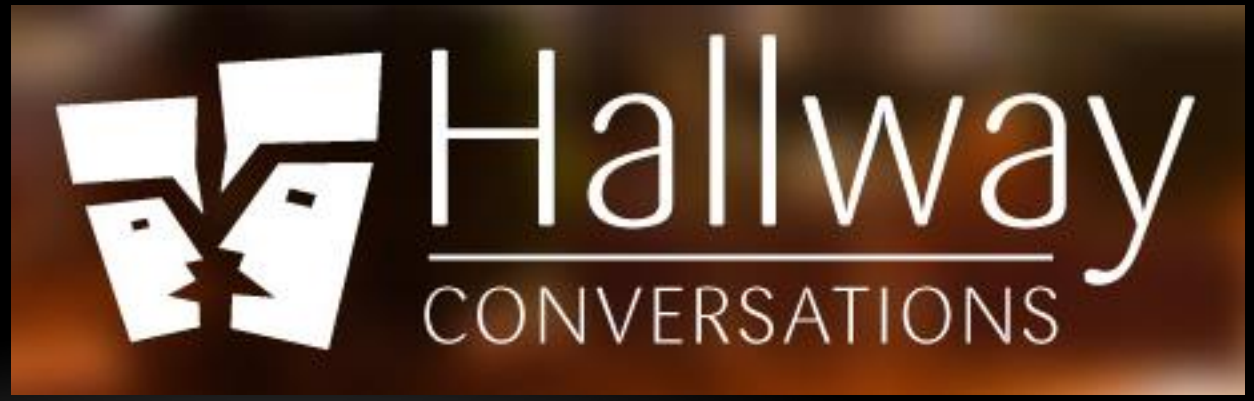
- Application Development and Application Renovations
  - Architecture, Development, and Architectural Design Reviews
  - .NET, Java, SharePoint
- Agility - Agile Coaching and Transformations
- Cloud Enablement
- Mobile – iOS, Android, Windows
- Training - Agile, .NET
- Our place or yours!
- Contact me at phil@sds-consulting.com

Some of our Customers (That I can Share)

The David J. Joseph Company

HALLWAY CONVERSATIONS PODCAST

- Hosted by Phil Japikse, Steve Bohlen, Lee Brandt, James Bender
- Website: www.hallwayconversations.com
- iTunes: http://bit.ly/hallway_convo_itunes
- Feed Burner: http://bit.ly/hallway_convo_feed
- Also available through Windows Store



WHAT IS ENTITY FRAMEWORK?

- Entity Framework (EF) is an object-relational mapper that enables .NET developers to work with relational data using domain-specific objects. It eliminates the need for most of the data-access code that developers usually need to write.
- Getting Entity Framework 6.x
  - For your project - get it from NuGet
    - http://nuget.org/packages/EntityFramework/
    - PM> Install-Package EntityFramework
- For Visual Studio
  - VS 2013 – It's included
  - VS 2012 – https://www.microsoft.com/en-us/download/details.aspx?id=40762

THE DATABASE CONTEXT

- Defines tables as DbSets
- Derives from DbContext
  - Constructor passes in connection string to the base class
  - DbContext class provides the meat of the functionality
- Provides ObjectMaterialized event for processing reconstituted classes
- Provides SavingChanges event for processing while data is persisted to data source

A TALE OF TWO CITIES

- Database Designer/EDMX
  - Visual representation of database tables
  - Code classes are (re)created during the build process
  - GOING AWAY IN EF7!
- Code First
  - Code representation of database tables
  - Uses data annotations to classes
- Regardless of choosing designer or code first, you can start from existing database or no database

USING THE EF DESIGNER

- Use an edmx file to hold the visual tables and relationships
  - Context name is set in the designer
- Tables are created and modified directly in the designer
  - Properties (tables/fields) are set in the designer
- Classes are created by [edmxFileName].tt – (t4 template)
  - Classes are located under this template in solution explorer
  - Classes are created very plainly
- Can reverse engineer an existing database
  - Note: HeirarchyId is not supported

ADDING DATA ANNOTATIONS TO DESIGNER GENERATED CLASSES

- Create partial class
- Add MetadataType(typeof(<class that holds the annotations>) attribute
- In metadata class, add annotations to public fields

```
[MetadataType(typeof(Address))]
public class AddressMetadata
{
  [StringLength(150)]
  [Display(Name="Address Line 1")]
  public string AddressLine1
     { get; set; }
}
```

USING CODE FIRST

- Tables are defined by classes
  - Add DbSet<type> to Context for each table
- Fields are defined by properties
  - Restrictions/Rules set with
    - Data Annotations
    - Fluent API in OnModelCreating
- Relations are defined by collections (ICollection)
- Changes are handled with Migrations

ADDING ENTITY FRAMEWORK CONTEXT

- Make sure Entity Framework is added to the project using NuGet
- Add -> New Item -> ADO.NET Entity Data Model
- Four options
  - EF Designer from Database
  - Empty EF Designer Model
  - Empty Code First Model
  - Code First from database
- Specify Db Connection and name (as it will appear in the app/web.config)

CODE FIRST TABLES/PROPERTIES

CREATING CLASSES AND PROPERTIES

- Create Class per table
  - Add Table attribute to change schema/name (or use Fluent API)
  - Add DbSet<Type> to Context
- Create Properties
  - Primary Key
  - Rowversion – mark with [Timestamp] attribute
- Create Relations

```
public class Category
{
    [Key]
    public int Id { get; set; }
    [StringLength(100),
                Display(Name="Category Name")]
    public string CategoryName { get; set; }
    [Timestamp]
    public byte[] TimeStamp { get; set; }
    public virtual ICollection<Product>
                Products { get; set; }
}
-----------------------------------------
public partial class StoreCatalogContext :
DbContext
{
    public StoreCatalogContext()
       :base("name=StoreCatalog") {}
    public virtual DbSet<Category> Categories
       { get; set; }
}
```

FREQUENTLY USED ATTRIBUTES FOR FIELDS

- Key – sets the primary key
  - Optional if field is named Id or <Type>Id
  - Defaults to Identity - use DatabaseGenerated(DatabaseGeneratedOption)
    - Identity, Computed, or None
- Timestamp – use on byte[] type to create RowVersion
- Required – sets field as non-nullable
- StringLength – sets length on nvarchar fields
- DefaultValue
- DataType(DataType.Text || DataType.Date)
  - Used to set hints on field creation - many more types to choose from

CREATING RELATIONS

- Parent
  - Add virtual ICollection<Type>
- Child
  - Add virtual <Type>
    - Specify [ForeignKey("fieldname")]
      - Not needed if following the standard naming style
    - Add <Type>Id
      - If not nullable, will cascade delete
      - Can set cascade options via Fluent API

```
public class Category
{
    [Key]
    public int Id { get; set; }
    public virtual ICollection<Product>
      Products { get; set; }
}

public partial class Product
{
    [Key]
    public int Id { get; set; }
    [Required]
    public int CategoryId { get; set; }
    [ForeignKey("CategoryId")]
    public virtual Category Category
       { get; set; }
}
```

USING THE FLUENT API

- Must use to set precision on fields
- Use to set cascade delete options
  - Much clearer than the obscure rules based on nullability and required attributes
  - Must use it to set cascade on one to one relationships

```
modelBuilder.Entity<Product>()
   .ToTable("Products", "Catalog");
modelBuilder.Entity<ProductPhoto>()
   .ToTable("ProductPhotos", "Catalog");

modelBuilder.Entity<Product>()
   .Property(x => x.CurrentPrice)
   .HasPrecision(10, 4);

modelBuilder.Entity<Product>()
   .HasOptional(x => x.Image)
   .WithRequired(x => x.ProductParent)
   .WillCascadeOnDelete(true);
```

LOGGING INTERCEPTION (BUILT-IN)

- EF 6.1+ has built in logging
- Add Interceptor into config file
- Two parameters
  - File location and name
  - Append to file (default is overwrite)

```
<interceptors>
  <interceptor
type="System.Data.Entity.Infrastructure.
Interception.DatabaseLogger,EntityFramew
ork">
   <parameters>
    <parameter
value="c:\Temp\LogOutput.txt"/>
    <parameter value="true"
type="System.Boolean"/>
   </parameters>
  </interceptor>
</interceptors>
```

CUSTOM INTERCEPTORS

- Inherit from IDbCommandInterceptor
- Register in App.Config
- Largely replaced with DbContext Events ObjectMaterialized and SavingChanges

```
public class LoggingInterceptor : IDbCommandInterceptor
{
    public void NonQueryExecuting(DbCommand command,
    DbCommandInterceptionContext<int> context) {}

    public void NonQueryExecuted(DbCommand command,
    DbCommandInterceptionContext<int> context) {}

    public void ReaderExecuting(DbCommand command,
    DbCommandInterceptionContext<DbDataReader> context) {}

    public void ReaderExecuted(DbCommand command,
    DbCommandInterceptionContext<DbDataReader> context) {}

    public void ScalarExecuting(DbCommand command,
    DbCommandInterceptionContext<object> context) {}

    public void ScalarExecuted(DbCommand command,
    DbCommandInterceptionContext<object> context) {}
}
-----------------------------------------------
<interceptor
type="EF_Examples_DAL.Context.LoggingInterceptor, EF-
Examples-DAL">
</interceptor>
```

HANDLING THE DBCONTEXT EVENTS

- Create a parameterized ctor in partial class for the Context
  - Make sure to call the default ctor!
- Get the ObjectContext
  - Must cast context to IObjectContextAdapter
  - Wire up the events

```
public partial class ProductContext
{
  private ObjectContext _objectContext;

  public ProductContext(bool useEvents)
    :this()
  {
     if (!useEvent) return;

     _objectContext =
       (this as IObjectContextAdapter)
       .ObjectContext;

     _objectContext.ObjectMaterialized +=
       OnObjectMaterialized;

     _objectContext.SavingChanges +=
       OnSavingChanges;
  }
}
```

HANDLING OBJECTMATERIALIZED EVENT

- Create a parameterized ctor in partial class for the Context
- Get the ObjectContext
  - Must cast context to IObjectContextAdapter
- Handle the ObjectMaterialized event
  - Use the Entity property of the event args

```
public partial class ProductContext
{
  private void OnObjectMaterialized(
    object sender,
    ObjectMaterializedEventArgs e)
  {
    if (e.Entity is ModelBase)
    {
       ((ModelBase)e.Entity)
         .IsChanged = false;
    }
  }
}
```

HANDLING SAVINGCHANGES EVENT

- Fires just before data is persisted to the database
- Allows for modification, validation, etc. prior to saving
  - Can reset properties to original values
  - Can cancel save process

```
void OnSavingChanges(
  object sender, EventArgs e)
{
  var context = sender as ObjectContext;
  foreach (ObjectStateEntry item in
    context.ObjectStateManager.GetObjectStateEntries(
      EntityState.Modified))
  {
     if (!item.IsRelationship && (item.Entity.GetType() ==
        typeof(ProductPhoto)))
     {
       var photo = item.Entity as ProductPhoto;

       var modifiedProperties =
          item.GetModifiedProperties().ToList();
       foreach (var prop in modifiedProperties)
       {
         item.RejectPropertyChanges(prop);
       }
     }
  }
}
```

DATA INITIALIZERS

DATA INITIALIZERS

- Used to add data to the database for testing
- Can be set to reload data every run or on model changes
- Can be called from code or set in config file

```
public class SampleDataInitializer :
   DropCreateDatabaseAlways<StoreCatalogContext>
{
   protected override void Seed(
      StoreCatalogContext context)
   {
       var cats = new List<Category>
       {
         new Category {CategoryName =
"Communications",Products = GetProducts(1)},
                new Category {CategoryName =
"Deception",Products = GetProducts(2)},
        };
        cats.ForEach(x =>
            context.Categories.Add(x));
        context.SaveChanges();
   }
}

Database.SetInitializer<StoreCatalogContext>(
   new SampleDataInitializer());
```

CRUD OPERATIONS

LOAD DATA FROM DATABASE

- Create new instance of your Context (e.g. ProductContext)
  - Tables are exposed as properties on the context
- Use LINQ to get data from the tables
- Remember when LINQ executes
  - Use ToList(), First(), etc. where appropriate

```
public Product GetOne(int id)
{
  return _db.Products
    .FirstOrDefault(
       x => x.ProductID == id);
}
public List<Product> GetAll()
{
  return _db.Products.ToList();
}
public Product GetFirst()
{
  return _db.Products
   .FirstOrDefault();
}
```

PERFORM EAGER FETCH

- By default, EF performs lazy loading of related entities
  - Can force eager fetching per query
- Can change default to be eager (use with caution!)
- To turn off lazy loading for a particular property, do not make it virtual.

```
public List<Category>
GetAllWithProducts()
{
    return Context
      .Categories
      .Include(x => x.Products)
      .ToList();
}
```

UPDATE DATA RECORD

- Must retrieve object from Context
- Update fields
- Call SaveChanges

```
ProductPhoto photo = _productContext
      .ProductPhotoes
      .FirstOrDefault(x =>
       x.ProductPhotoID == _photoId);
photo.ThumbnailPhotoFileName =
"Updated";
_productContext.SaveChanges();
```

CONCURRENCY CHECKING

- User Timestamp fields
- Catch DbUpdateConcurrencyException
- All affected Entities are in the Entries property of the exception
- Each entity exposes current, database, and original values

```
var entity = dbce.Entries.First();
var serverCategory =
  (Category)entity.GetDatabaseValues()
  .ToObject();
var clientCategory =
  (Category)entity.CurrentValues
  .ToObject();

Console.WriteLine(message,"Category Name: ",
   serverCategory.CategoryName,
   clientCategory.CategoryName);
Console.WriteLine(message, "Rowversion:",
   serverCategory.TimeStamp[7],
   clientCategory.TimeStamp[7]);
```

ADD NEW RECORD

- Instantiate a new object
  - Set the properties
- Call Add on DbSet
- Call SaveChanges
- Primary Key is populated by EF*

```
private ProductContext
_productContext;
private ProductPhoto _photo;
_productContext = new
ProductContext();
_photo = new ProductPhoto
{
   ThumbnailPhotoFileName =
"123456789022",
   ModifiedDate = DateTime.Now
};
_productContext.ProductPhotoes.Add(_p
hoto);
_productContext.SaveChanges();
```

DELETE A RECORD

- Set entity state to deleted
- OR
- Call Remove on DbSet
  - Must have instance of object to delete

- Call SaveChanges

```
_productContext.ProductPhotos
   .Remove(_photo);
Context.SaveChanges();

Context.Entry(entity).State =
   EntityState.Deleted;

Context.SaveChangesAsync();
```

MIGRATIONS

EF MIGRATIONS

- Used to modify schema of database already deployed/created
  - Can also seed data
- EF 6.x now supports more than one DbContext
  - E.g. ApplicationDbContext (ASP.NET Identity) and MyDomainModelContext
- Three steps
  - Enable Migrations
  - Add migration
    - Can be set to automatically migrate
  - Update database

ENABLE MIGRATIONS

- Open Package Manager Console
  - Enable-Migrations [–ContextTypeName <fully qualified name>] [-MigrationsDirectory <directoryname>] [-projectname <projectname>] [-force]
- Example:
  - Enable-Migrations -ContextTypeName EF_Examples_DAL.Context.StoreCatalogContext -MigrationsDirectory EF\Migrations\AppData -ProjectName EF-Examples-DAL
- Creates Directory for Migrations,
  - [TimeStamp]_InitialCreate and Configuration files

THE MIGRATION FILES

- [TimeStamp]_[MigrationName]
  - Migration name == InitialCreate for the first one
  - Derives from DbMigration
  - Up method creates/Updates the database
  - Down unwinds any changes
- Configuration
  - Specifies Directory, Context, and automatic migrations setting
  - Also has the Seed method to add or update data in the database

ADDING ADDITIONAL MIGRATIONS

- Open Package Manager Console
  - Add-Migrations [-Name] <string> [–ConfigurationTypeName <fully qualified name>] [-projectname <projectname>] [-force]
- Example:
  - add-migration ParentCategory -configurationtypename EF_Examples_DAL.EF.Migrations.AppData.Configuration -projectname EF-Examples-DAL -force
- Creates [TimeStamp]_ParentCategory

UPDATING THE DATABASE

- Open Package Manager Console
  - Update-database [-TargetMigration <Name>] [–ConfigurationTypeName <fully qualified name>] [-projectname <projectname>] [-ConectionStringName <connectionString]
- Example:
  - Update-Database –targetmigration InitialCreate -configurationtypename EF_Examples_DAL.EF.Migrations.AppData.Configuration -projectname EF-Examples-DAL -force

DEMO

Entity Framework

Questions?

Contact Me

phil@sds-consulting.com
www.sds-consulting.com

skimedic@outlook.com
www.skimedic.com/blog
www.twitter.com/skimedic
www.hallwayconversations.com

www.about.me/skimedic

Thank You!