



Module "Data processing"

Submodule "Data processing technologies"

ADO.NET Entity Framework Pt.2

Contents

- Querying data
- Data Loading: lazy vs eager
- LINQ to Entities
- Data Annotations
- Fluent Configuration
- Migrations
- Repository and UoW patterns

Querying and Finding Entities

Finding entities using a Query

`DbSet` and `IdbSet` implement `IQueryable` and so can be used as the starting point for writing a LINQ query against the database:

```
using (var context = new BloggingContext())
{
    // Query for all blogs with names starting with B
    var blogs = from b in context.Blogs
                where b.Name.StartsWith("B")
                select b;

    // Query for the Blog named ADO.NET Blog
    var blog = context.Blogs
                .Where(b => b.Name == "ADO.NET Blog")
                .FirstOrDefault();
}
```

Finding entities using a Query

A query is executed against the database when:

- it is enumerated by a **foreach** statement.
- It is enumerated by a collection operation such as **ToArray**, **ToDictionary**, or **ToList**.
- LINQ operators such as **First** or **Any** are specified in the outermost part of the query.
- The following methods are called: the **Load** extension method on a **DbSet**, **DbEntityEntry.Reload**, and **Database.ExecuteNonQueryCommand**.

Finding entities using Primary Keys

The **Find** method on **DbSet** uses the primary key value to attempt to find an entity tracked by the context:

```
using (var context = new BloggingContext())
{
    // Will hit the database
    var blog = context.Blogs.Find(3);

    // Will return the same instance without hitting the database
    var blogAgain = context.Blogs.Find(3);

    context.Blogs.Add(new Blog { BlogId = -1 });

    // Will find the new blog even though it does not exist in the database
    var newBlog = context.Blogs.Find(-1);

    // Will find a User which has a string primary key
    var user = context.Users.Find("johndoe1987");
}
```

Finding entities using Primary Keys

Find is different from using a query in two significant ways:

- A round-trip to the database will only be made if the entity with the given key is not found in the context.
- Find will return entities that are in the **Added** state.

Finding an entity by Composite Primary Key

Find method overload allows to find entities by more than one property (i.e. composite key):

```
using (var context = new BloggingContext())
{
    var settings = context.BlogSettings.Find(3, "johndoe1987");
}
```

Note: When you have composite keys you need to use **ColumnAttribute** or the fluent API to specify an ordering for the properties of the composite key. The call to **Find** must use this order when specifying the values that form the key.

The Load Method

There are several scenarios where you may want to load entities from the database into the context without immediately doing anything with those entities.

Example: Load is used in Windows Forms data binding application

```
protected override void OnLoad(EventArgs e)
{
    base.OnLoad(e);

    _context = new ProductContext();

    _context.Categories.Load();
    categoryBindingSource.DataSource = _context.Categories.Local.ToBindingList();
}
```

The Load Method

Using **Load** method to load a filtered collection of related entities:

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);

    // Load the posts with the 'entity-framework' tag related to a given blog
    context.Entry(blog)
        .Collection(b => b.Posts)
        .Query()
        .Where(p => p.Tags.Contains("entity-framework"))
        .Load();
}
```

No-Tracking Queries

`AsNoTracking` extension method allows getting entities back from a query but not having those entities tracked by the context:

```
private static void NoTracking()
{
    using (var context = new BloggingContext())
    {
        // Query for all blogs without tracking them
        var blogs1 = context.Blogs.AsNoTracking();

        // Query for some blogs without tracking them
        var blogs2 = context.Blogs
            .Where(b => b.Name.Contains(".NET"))
            .AsNoTracking()
            .ToList();
    }
}
```

Writing SQL queries for entities

The `SqlQuery` method on `DbSet` allows a raw SQL query to be written that will return entity instances:

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs.SqlQuery("SELECT * FROM dbo.Blogs").ToList();
}

using (var context = new BloggingContext())
{
    var result = context.Blogs.SqlQuery(
        "SELECT * FROM Blogs WHERE BlogId=@id",
        new SqlParameter("@id", 1)).FirstOrDefault();

    Console.WriteLine(result.Name);
    Console.ReadLine();
}
```

Loading entities from stored procedures

You can use `DbSet.SqlQuery` to load entities from the results of a stored procedure:

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs.SqlQuery("dbo.GetBlogs").ToList();
}
```

```
using (var context = new BloggingContext())
{
    int blogId = 1;

    var blogs = context.Blogs.SqlQuery("dbo.GetBlogById @p0", blogId).Single();
}
```

Writing SQL queries for non-entity types

A SQL query returning instances of any type, including primitive types, can be created using the `SqlQuery` method on the `Database` class:

```
using (var context = new BloggingContext())
{
    var blogNames = context.Database.SqlQuery<string>(
        "SELECT Name FROM dbo.Blogs").ToList();
}
```

Sending raw commands to the database

Non-query commands can be sent to the database using the `ExecuteSqlCommand` method on `Database`:

```
using (var context = new BloggingContext())
{
    context.Database.ExecuteSqlCommand(
        "UPDATE dbo.Blogs SET Name = 'Another Name' WHERE BlogId = 1");
}
```

Loading Related Entities

Entity Framework supports three ways to load related data:

- Eager loading
- Lazy loading
- Explicit loading

Eagerly Loading

Eager loading is the process whereby a query for one type of entity also loads related entities as part of the query:

```
using (var context = new BloggingContext())
{
    var blogs1 = context.Blogs
        .Include(b => b.Posts)
        .ToList();

    var blog1 = context.Blogs
        .Where(b => b.Name == "ADO.NET Blog")
        .Include(b => b.Posts)
        .FirstOrDefault();

    var blogs2 = context.Blogs
        .Include("Posts")
        .ToList();

    var blog2 = context.Blogs
        .Where(b => b.Name == "ADO.NET Blog")
        .Include("Posts")
        .FirstOrDefault();
}
```

Load all blogs and related posts.

Load one blog and its related posts

Load all blogs and related posts, using a string to specify the relationship.

Load one blog and its related posts, using a string to specify the relationship.

Loading multiple levels

Eager loading is the process whereby a query for one type of entity also loads related entities as part of the query:

```
using (var context = new BloggingContext())
{
    var blogs1 = context.Blogs
        .Include(b => b.Posts.Select(p => p.Comments))
        .ToList();

    var users1 = context.Users
        .Include(u => u.Profile.Avatar)
        .ToList();

    var blogs2 = context.Blogs
        .Include("Posts.Comments")
        .ToList();

    var users2 = context.Users
        .Include("Profile.Avatar")
        .ToList();
}
```

Load all blogs, all related posts, and all related comments

Load all users, their related profiles, and related avatar

Load all blogs, all related posts, and all related comments using a string to specify the relationships

Load all users, their related profiles, and related avatar using a string to specify the relationships

Lazy Loading

Lazy loading is the process whereby an entity or collection of entities is automatically loaded from the database the first time that a property referring to the entity/entities is accessed

```
public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }
    public string Tags { get; set; }

    public virtual ICollection<Post> Posts { get; set; }
}
```

Turn lazy loading off for serialization

Lazy loading and serialization don't mix well, and if you aren't careful you can end up querying for your entire database just because lazy loading is enabled.

Most serializers work by accessing each property on an instance of a type.

Property access triggers lazy loading, so more entities get serialized.

On those entities properties are accessed, and even more entities are loaded.

It's a good practice to turn lazy loading off before you serialize an entity. The following sections show how to do this.

Turning off lazy loading for specific navigation properties

Lazy loading of the **Posts** collection can be turned off by making the **Posts** property *non-virtual*:

```
public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }
    public string Tags { get; set; }

    public ICollection<Post> Posts { get; set; }
}
```

Turn off lazy loading for all entities

Lazy loading can be turned off for all entities in the context by setting a flag on the `Configuration` property:

```
public class BloggingContext : DbContext
{
    public BloggingContext()
    {
        this.Configuration.LazyLoadingEnabled = false;
    }
}
```

Explicitly Loading

Even with lazy loading disabled it is still possible to lazily load related entities, but it must be done with an explicit call:

```
using (var context = new BloggingContext())
{
    var post = context.Posts.Find(2);

    context.Entry(post).Reference(p => p.Blog).Load();

    context.Entry(post).Reference("Blog").Load();

    var blog = context.Blogs.Find(1);

    context.Entry(blog).Collection(p => p.Posts).Load();

    context.Entry(blog).Collection("Posts").Load();
}
```

Load the blog related
to a given post

Load the blog related to a given post
using a string

Load the posts related
to a given blog

Load the posts related to a
given blog using a string to
specify the relationship

Applying filters when explicitly loading related entities

```
**  
using (var context = new BloggingContext())  
{  
    var blog = context.Blogs.Find(1);  
  
    // Load the posts with the 'entity-framework' tag related to a given blog.  
    context.Entry(blog)  
        .Collection(b => b.Posts)  
        .Query()  
        .Where(p => p.Tags.Contains("entity-framework"))  
        .Load();  
  
    // Load the posts with the 'entity-framework' tag related to a given blog  
    // using a string to specify the relationship.  
    context.Entry(blog)  
        .Collection("Posts")  
        .Query()  
        .Where(p => p.Tags.Contains("entity-framework"))  
        .Load();  
}
```


Using Query to count related entities without loading them

Sometimes it is useful to know how many entities are related to another entity in the database without actually incurring the cost of loading all those entities. The **Query** method with the LINQ Count method can be used to do this:

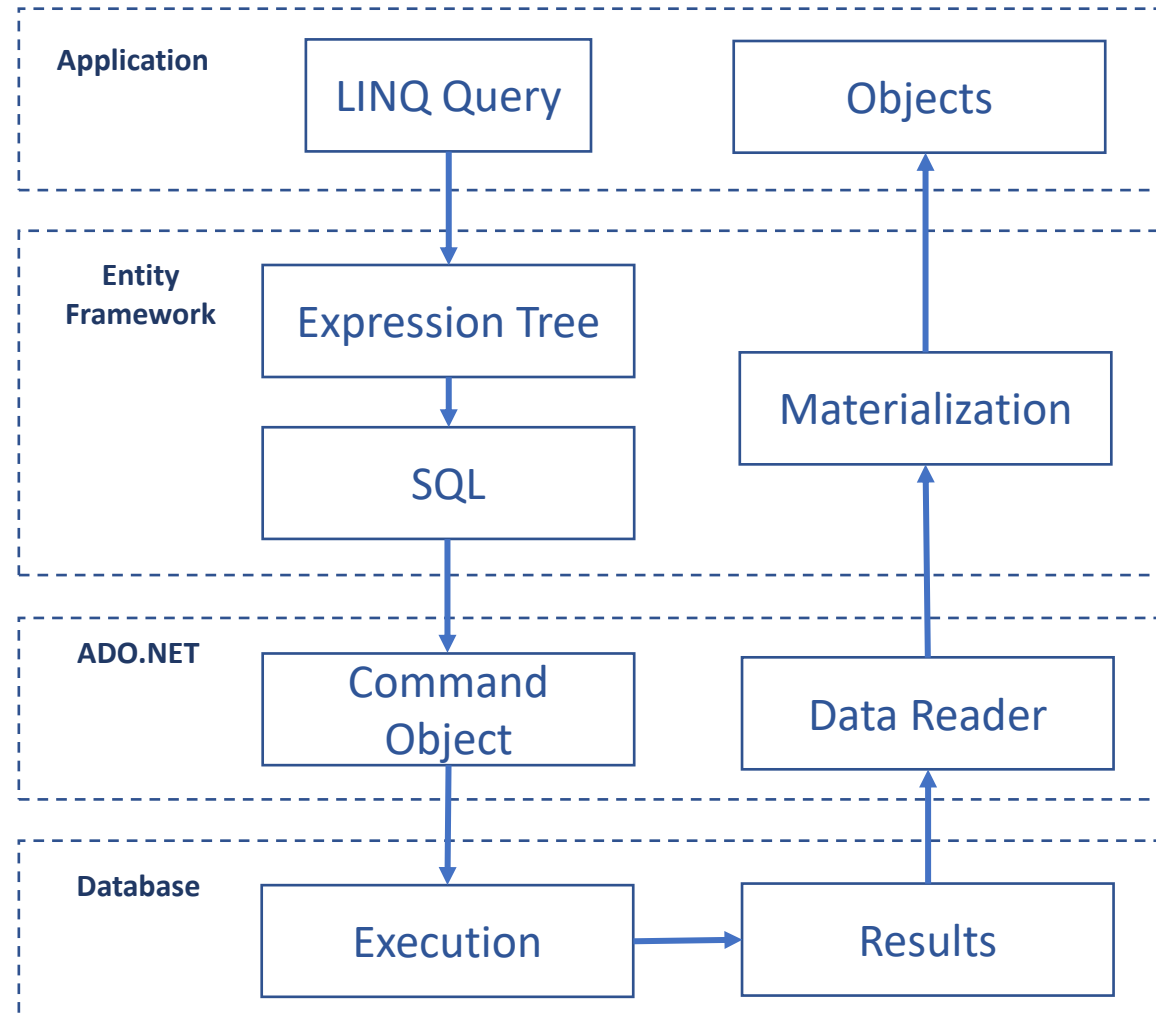
```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);

    // Count how many posts the blog has.
    var postCount = context.Entry(blog)
                           .Collection(b => b.Posts)
                           .Query()
                           .Count();
}
```

LINQ to Entities

Query Execution

**



Deferred query execution

Deferred execution enables multiple queries to be combined or a query to be extended. When a query is extended, it is modified to include the new operations, and the eventual execution will reflect the changes:

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Product> productsQuery =
        from p in context.Products
        select p;

    IQueryable<Product> largeProducts = productsQuery.Where(p => p.Size == "L");

    Console.WriteLine("Products of size 'L':");
    foreach (var product in largeProducts)
    {
        Console.WriteLine(product.Name);
    }
}
```

Immediate Query Execution

In contrast to the deferred execution of queries that produce a sequence of values, queries that return a singleton value are executed immediately.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Product> products = context.Products;

    Product[] prodArray = (
        from product in products
        orderby product.ListPrice descending
        select product).ToArray();

    Console.WriteLine("Every price from highest to lowest:");
    foreach (Product product in prodArray)
    {
        Console.WriteLine(product.ListPrice);
    }
}
```

Store Execution

- Expressions in LINQ to Entities are evaluated on the server, and the behavior of the expression should not be expected to follow common language runtime (CLR) semantics, but those of the data source.
- Some expressions in the query might be executed on the client. In general, most query execution is expected to occur on the server.
- Certain operations are always executed on the client, such as binding of values, sub expressions, sub queries from closures, and materialization of objects into query results.

Method-Based Projection

Use of the **Select** method to project the **Product.Name** and **Product.ProductID** properties into a sequence of anonymous types:

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query = context.Products
        .Select(product => new
        {
            ProductId = product.ProductID,
            ProductName = product.Name
        });

    Console.WriteLine("Product Info:");
    foreach (var productInfo in query)
    {
        Console.WriteLine("Product Id: {0} Product name: {1} ",
            productInfo.ProductId, productInfo.ProductName);
    }
}
```

Method-Based Filtering

Example returns the orders where the order quantity is greater than 2 and less than 6:

```
int orderQtyMin = 2;
int orderQtyMax = 6;
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query = context.SalesOrderDetails
        .Where(order => order.OrderQty > orderQtyMin && order.OrderQty < orderQtyMax)
        .Select(s => new { s.SalesOrderID, s.OrderQty });

    foreach (var order in query)
    {
        Console.WriteLine("Order ID: {0} Order quantity: {1}",
            order.SalesOrderID, order.OrderQty);
    }
}
```


Method-Based Ordering

Use of **OrderBy** and **ThenBy** to return a list of contacts ordered by last name and then by first name:

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Contact> sortedContacts = context.Contacts
        .OrderBy(c => c.LastName)
        .ThenBy(c => c.FirstName);

    Console.WriteLine("The list of contacts sorted by last name then by first name:");
    foreach (Contact sortedContact in sortedContacts)
    {
        Console.WriteLine(sortedContact.LastName + ", " + sortedContact.FirstName);
    }
}
```

Mixed Syntax Aggregate Operators

Use of the Average method to find the average list price of the products of each style:

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Product> products = context.Products;

    var query = from product in products
                group product by product.Style into g
                select new
                {
                    Style = g.Key,
                    AverageListPrice =
                        g.Average(product => product.ListPrice)
                };

    foreach (var product in query)
    {
        Console.WriteLine("Product style: {0} Average list price: {1}",
            product.Style, product.AverageListPrice);
    }
}
```

Method-Based Join Operators

The following example performs a **GroupJoin** over the **SalesOrderHeader** and **SalesOrderDetail** tables to find the number of orders per customer:

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;
    ObjectSet<SalesOrderDetail> details = context.SalesOrderDetails;

    var query = orders.GroupJoin(details,
        order => order.SalesOrderID,
        detail => detail.SalesOrderID,
        (order, orderGroup) => new
        {
            CustomerID = order.SalesOrderID,
            OrderCount = orderGroup.Count()
        });

    foreach (var order in query)
    {
        Console.WriteLine("CustomerID: {0}  Orders Count: {1}",
            order.CustomerID,
            order.OrderCount);
    }
}
```

Query Expression Grouping

Returns **Address** objects grouped by postal code. The results are projected into an anonymous type:

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query =
        from address in context.Addresses
        group address by address.PostalCode into addressGroup
        select new
        {
            PostalCode = addressGroup.Key,
            AddressLine = addressGroup

        };

    foreach (var addressGroup in query)
    {
        Console.WriteLine("Postal Code: {0}", addressGroup.PostalCode);
        foreach (var address in addressGroup.AddressLine)
        {
            Console.WriteLine("\t" + address.AddressLine1 +
                               address.AddressLine2);
        }
    }
}
```

Query Expression Navigating Relationships

Example uses **Select** to get all contact IDs and the *sum* of the total due for each contact by specified last name:

```
string lastName = "Doe";
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;

    var ordersQuery = from contact in contacts
                      where contact.LastName == lastName
                      select new
                      {
                          ContactID = contact.ContactID,
                          Total = contact.SalesOrderHeaders.Sum(o => o.TotalDue)
                      };
}
```

Query Expression Navigating Relationships

Example uses **Select** to get all contact IDs and the *sum* of the total due for each contact by specified last name:

```
string lastName = "Doe";
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;

    var ordersQuery = from contact in contacts
                      where contact.LastName == lastName
                      select new
                      {
                          ContactID = contact.ContactID,
                          Total = contact.SalesOrderHeaders.Sum(o => o.TotalDue)
                      };
}
```

Database Functions

The example calls the aggregate **ChecksumAggregate** method directly. Note that an **ObjectQuery<T>** is passed to the function, which allows it to be called without being part of a LINQ to Entities query.

```
using (AdventureWorksEntities AWEntities = new AdventureWorksEntities())
{
    // SqlFunctions.ChecksumAggregate is executed in the database.
    decimal? checkSum = SqlFunctions.ChecksumAggregate(
        from o in AWEntities.SalesOrderHeaders
        select o.SalesOrderID);

    Console.WriteLine(checkSum);
}
```

ObjectQuery<T>



Database Functions

The example executes a LINQ to Entities query that uses the **CharIndex** method to return all contacts with specified start characters of last name:

```
using (AdventureWorksEntities AWEntities = new AdventureWorksEntities())
{
    // SqlFunctions.CharIndex is executed in the database.
    var contacts = from c in AWEntities.Contacts
                   where SqlFunctions.CharIndex("Si", c.LastName) == 1
                   select c;

    foreach (var contact in contacts)
    {
        Console.WriteLine(contact.LastName);
    }
}
```


Code First Data Annotations

Data Annotations

The model:

```
public class Blog
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string BloggerName { get; set; }
    public virtual ICollection<Post> Posts { get; set; }
}

public class Post
{
    public int Id { get; set; }
    public string Title { get; set; }
    public DateTime DateCreated { get; set; }
    public string Content { get; set; }
    public int BlogId { get; set; }
    public ICollection<Comment> Comments { get; set; }
}
```

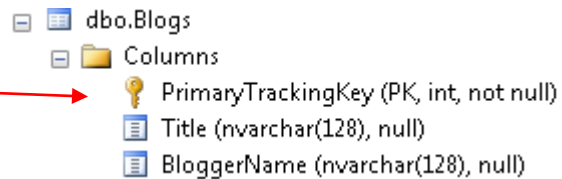
Key

Do not forget to include ComponentModel namespace:

```
using System.ComponentModel.DataAnnotations;
```

You can use the key annotation to specify which property is to be used as the EntityKey:

```
public class Blog
{
    [Key]
    public int PrimaryTrackingKey { get; set; }
    public string Title { get; set; }
    public string BloggerName { get; set; }
    public virtual ICollection<Post> Posts { get; set; }
}
```



Composite Keys

Entity Framework supports composite keys - primary keys that consist of more than one property:

```
public class Passport
{
    [Key]
    public int PassportNumber { get; set; }
    [Key]
    public string IssuingCountry { get; set; }
    public DateTime Issued { get; set; }
    public DateTime Expires { get; set; }
}
```

In fact attempting to use this class as is will result in an **InvalidOperationException**

Composite Keys

If you have entities with composite foreign keys, then you must specify the same column ordering that you used for the corresponding primary key properties

```
using System.ComponentModel.DataAnnotations;  
using System.ComponentModel.DataAnnotations.Schema;
```

```
public class Passport  
{  
    [Key]  
    [Column(Order = 1)]  
    public int PassportNumber { get; set; }  
    [Key]  
    [Column(Order = 2)]  
    public string IssuingCountry { get; set; }  
    public DateTime Issued { get; set; }  
    public DateTime Expires { get; set; }  
}
```

Composite Keys

If you have entities with composite foreign keys, then you must specify the same column ordering that you used for the corresponding primary key properties

```
using System.ComponentModel.DataAnnotations;  
using System.ComponentModel.DataAnnotations.Schema;
```

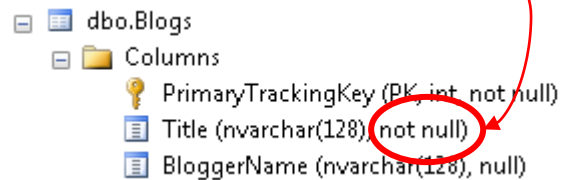
```
public class Passport  
{  
    [Key]  
    [Column(Order = 1)]  
    public int PassportNumber { get; set; }  
    [Key]  
    [Column(Order = 2)]  
    public string IssuingCountry { get; set; }  
    public DateTime Issued { get; set; }  
    public DateTime Expires { get; set; }  
}
```

Required

The Required annotation tells EF that a particular property is required

```
public class Blog
{
    [Required]
    public string Title { get; set; }

    //...
}
```



MaxLength and MinLength

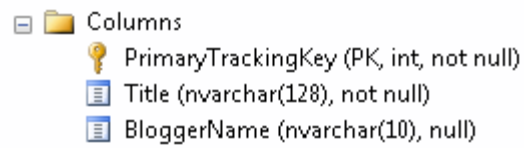
The **MaxLength** and **MinLength** attributes allow you to specify additional property validations, just as you did with Required

```
public class Blog
{
    //...

    [MaxLength(10), MinLength(5)]
    public string BloggerName { get; set; }

    //...
}
```

Only max value constraint will be added in database:



Error Messages

Many annotations let you specify an error message with the **ErrorMessage** attribute

```
public class Blog
{
    //...

    [MaxLength(10, ErrorMessage = "BloggerName must be 10 characters or less"), MinLength(5)]
    public string BloggerName { get; set; }

    //...
}
```

NotMapped

You can mark any properties that do not map to the database with the **NotMapped** annotation:

```
public class Blog
{
    [Required]
    public string Title { get; set; }

    [MaxLength(10), MinLength(5)]
    public string BloggerName { get; set; }

    [NotMapped]
    public string BlogCode
    {
        get
        {
            return Title.Substring(0, 1) + ":" + BloggerName.Substring(0, 1);
        }
    }
}
```

ComplexType

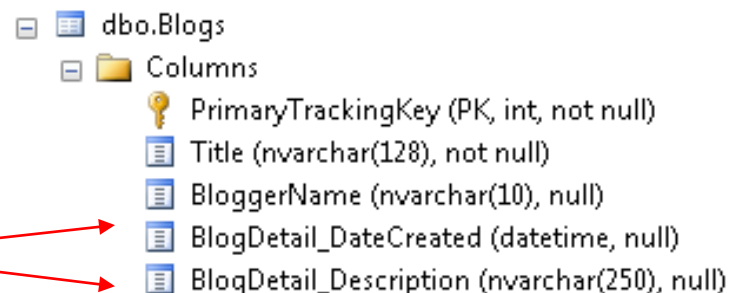
It's not uncommon to describe your domain entities across a set of classes and then layer those classes to describe a complete entity:

```
[ComplexType]
public class BlogDetails
{
    public DateTime? DateCreated { get; set; }

    [MaxLength(250)]
    public string Description { get; set; }
}
```

```
public class Blog
{
    public string Title { get; set; }
    public string BloggerName { get; set; }

    public BlogDetails BlogDetail { get; set; }
```



TimeStamp

EF will ensure that Timestamp field in database will be non-nullable. You can only have one timestamp property in a given class:

```
public class Blog
{
    [Timestamp]
    public Byte[] TimeStamp { get; set; }

    public string Title { get; set; }
    public string BloggerName { get; set; }

    public BlogDetails BlogDetail { get; set; }
}
```

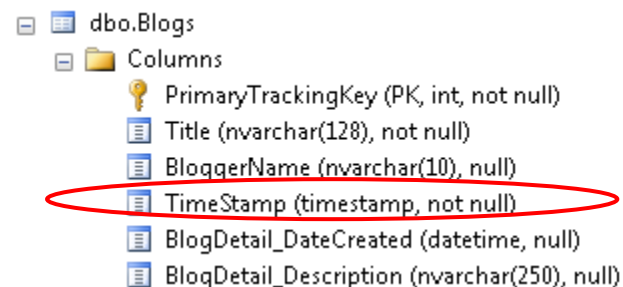
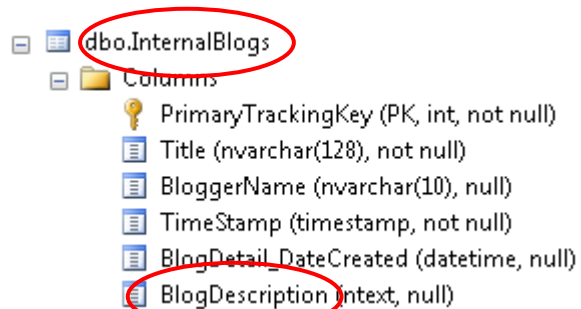


Table and Column

If you are letting Code First create the database, you may want to change the name of the tables and columns it is creating:

```
[Table("InternalBlogs")]
public class Blog
{
    [Column("BlogDescription", TypeName = "ntext")]
    public string Description { get; set; }
```



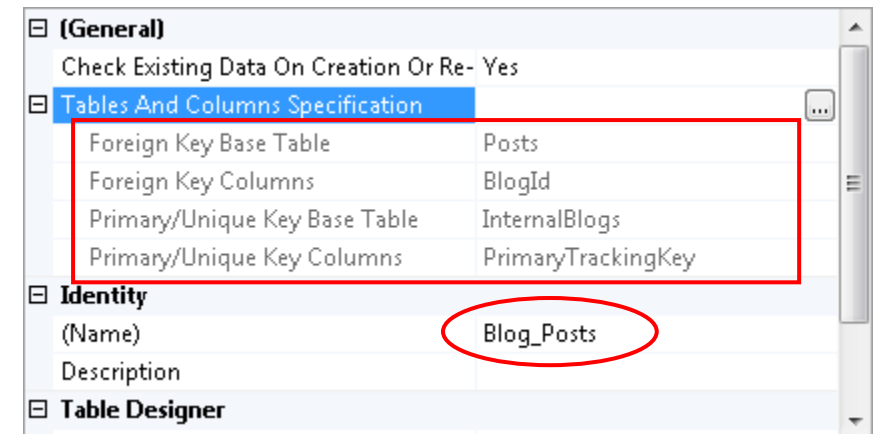
Relationship Attributes: ForeignKey

Foreign key recognized by convention, yet Blog's ID property may have different name, use ForeignKey annotation on navigation property to help Code First build the relationship:

```
public class Post
{
    public int Id { get; set; }
    public string Title { get; set; }
    public DateTime DateCreated { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }

    [ForeignKey("BlogId")]
    public Blog Blog { get; set; }
    public ICollection<Comment> Comments { get; set; }
}
```



Fluent API Configuration

Fluent API Configuration

You can use methods of `EntityTypeConfiguration` class to configure your entities relations and other attributes.

There are two ways to do it, first would be – declare your configuration in `OnModelCreating` method of context:

```
public class SchoolEntities : DbContext
{
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<OfficeAssignment>()
            .HasKey(t => t.InstructorID);
    }

    public DbSet<Course> Courses { get; set; }
    public DbSet<Department> Departments { get; set; }
    public DbSet<Instructor> Instructors { get; set; }
    public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
}
```


Fluent API Configuration

The second options would be to put your configurations into separate classes:

```
public class DepartmentConfiguration : EntityTypeConfiguration<Department>
{
    public DepartmentConfiguration()
    {
        this.HasKey(d => new { d.DepartmentID, d.Name });
        this.Property(d => d.Name).IsRequired();
        //etc.
    }
}

public class SchoolEntities : DbContext
{
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Configurations.Add(new DepartmentConfiguration());
    }
}
```

Configuring a One-to-“Zero-or-One”

The following example configures a one-to-zero-or-one relationship. The **OfficeAssignment** has the **InstructorID** property that is a primary key and a foreign key, because the name of the property does not follow the convention the **HasKey** method is used to configure the primary key:

```
// Configure the primary key for the OfficeAssignment
modelBuilder.Entity<OfficeAssignment>()
    .HasKey(t => t.InstructorID);

// Map one-to-zero or one relationship
modelBuilder.Entity<OfficeAssignment>()
    .HasRequired(t => t.Instructor)
    .WithOptional(t => t.OfficeAssignment);
```

Configuring a One-to-One

In most cases Entity Framework can infer which type is the dependent and which is the principal in a relationship. However, when both ends of the relationship are required or both sides are optional Entity Framework cannot identify the dependent and principal:

```
// Configure the primary key for the OfficeAssignment
modelBuilder.Entity<OfficeAssignment>()
    .HasKey(t => t.InstructorID);

modelBuilder.Entity<Instructor>()
    .IsRequired(t => t.OfficeAssignment)
    .WithRequiredPrincipal(t => t.Instructor);
```

Configuring a Many-to-Many

As is:

```
modelBuilder.Entity<Course>()  
    .HasMany(t => t.Instructors)  
    .WithMany(t => t.Courses);
```

If you want to specify the “join table” name and the names of the columns in the table you need to do additional configuration by using the **Map** method:

```
modelBuilder.Entity<Course>()  
    .HasMany(t => t.Instructors)  
    .WithMany(t => t.Courses)  
    .Map(m =>  
    {  
        m.ToTable("CourseInstructor");  
        m.MapLeftKey("CourseID");  
        m.MapRightKey("InstructorID");  
    });
```

Configuring a Relationship with One Navigation Property

By convention, Code First always interprets a unidirectional relationship as *one-to-many*.

For example, if you want a *one-to-one* relationship between `Instructor` and `OfficeAssignment`, where you have a navigation property on only the `Instructor` type, you need to use the fluent API to configure this relationship:

```
// Configure the primary Key for the OfficeAssignment
modelBuilder.Entity<OfficeAssignment>()
    .HasKey(t => t.InstructorID);

modelBuilder.Entity<Instructor>()
    .HasRequired(t => t.OfficeAssignment)
    .WithRequiredPrincipal();
```

Enabling Cascade Delete

You can configure cascade delete on a relationship by using the `WillCascadeOnDelete` method:

```
modelBuilder.Entity<Course>()  
    .HasRequired(t => t.Department)  
    .WithMany(t => t.Courses)  
    .HasForeignKey(d => d.DepartmentID)  
    .WillCascadeOnDelete(false);
```

You can remove these cascade delete conventions by using:

```
modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>();  
  
modelBuilder.Conventions.Remove<ManyToManyCascadeDeleteConvention>();
```

Configuring a Composite Foreign Key

If the primary key on the Department type consisted of DepartmentID and Name properties, you would configure the primary key for the Department and the foreign key on the Course types as follows:

```
// Composite primary key
modelBuilder.Entity<Department>()
    .HasKey(d => new { d.DepartmentID, d.Name });

// Composite foreign key
modelBuilder.Entity<Course>()
    .HasRequired(c => c.Department)
    .WithMany(d => d.Courses)
    .HasForeignKey(d => new { d.DepartmentID, d.DepartmentName });
```

Code First Migrations

Code First Migrations

Code First Migrations is the recommended way to evolve your application's database schema if you are using the Code First workflow. Migrations provide a set of tools that allow:

- Create an initial database that works with your EF model
- Generating migrations to keep track of changes you make to your EF model
- Keep your database up to date with those changes

Code First Migrations

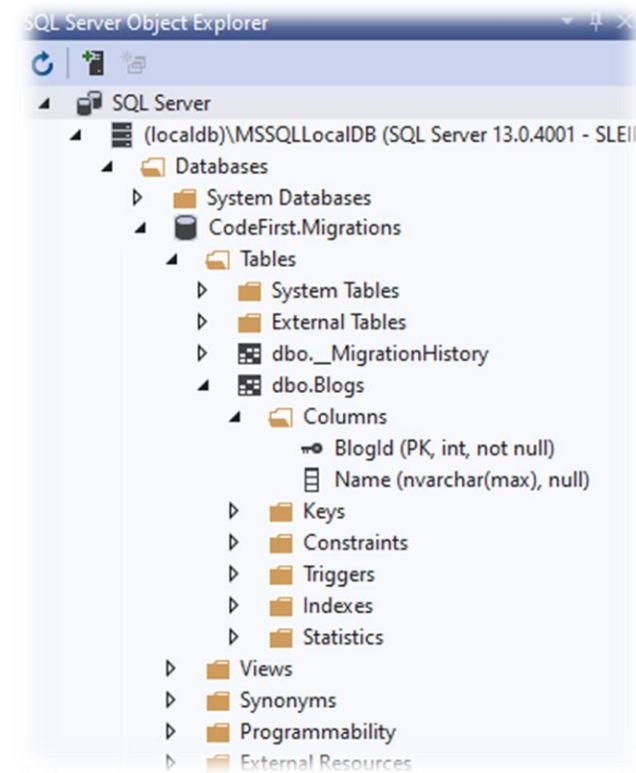
Before we start using migrations we need a Code First model.

```
public class BlogContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }
}

static void Main(string[] args)
{
    using (var db = new BlogContext())
    {
        db.Blogs.Add(new Blog { Name = "Another Blog " });
        db.SaveChanges();

        foreach (var blog in db.Blogs)
        {
            Console.WriteLine(blog.Name);
        }
    }
}
```



Code First Migrations

At some point we've decided to modify our entity:

```
public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }

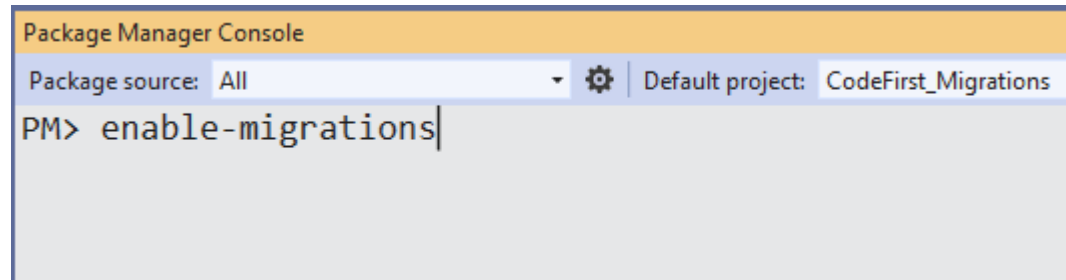
    public string Url { get; set; }
}
```

With the next application launch an exception will be raised, which hints on the next steps:



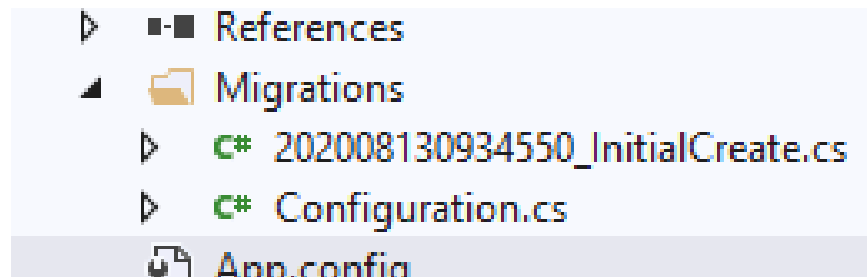
Code First Migrations

As the exception suggests, it's time to start using Code First Migrations. The first step is to enable migrations for our context:



```
Package Manager Console
Package source: All [v] [gear] Default project: CodeFirst_Migrations
PM> enable-migrations
```

Which will add Migrations folder to project:



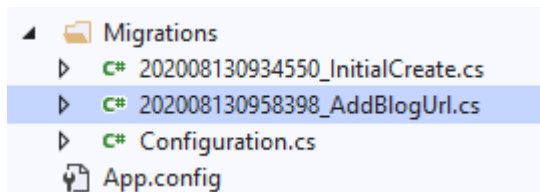
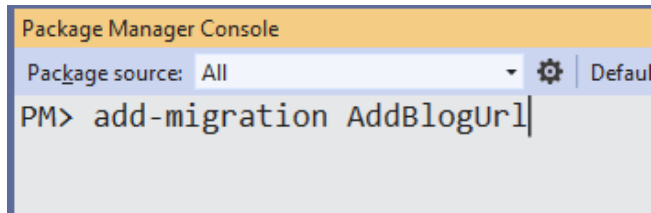
Code First Migrations

Code First Migrations has two primary commands that you are going to become familiar with:

- **Add-Migration** will scaffold the next migration based on changes you have made to your model since the last migration was created
- **Update-Database** will apply any pending migrations to the database

Code First Migrations

We need to scaffold a migration to take care of the new Url property we have added:



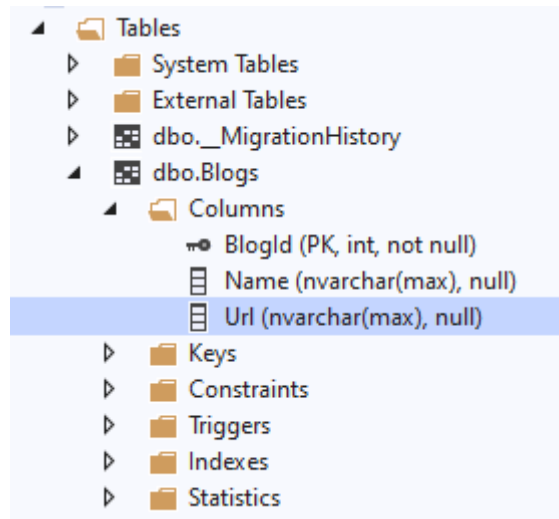
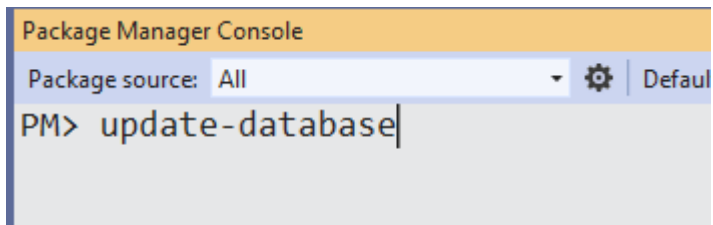
```
using System;
using System.Data.Entity.Migrations;

public partial class AddBlogUrl : DbMigration
{
    public override void Up()
    {
        AddColumn("dbo.Blogs", "Url", c => c.String());
    }

    public override void Down()
    {
        DropColumn("dbo.Blogs", "Url");
    }
}
```

Code First Migrations

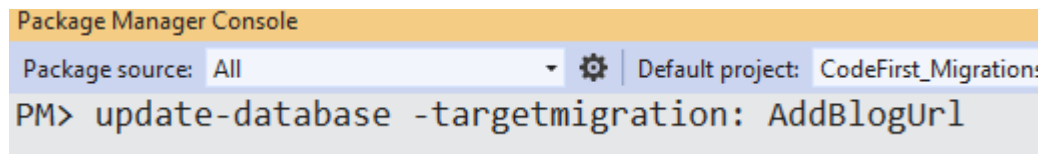
Run Update-Database and Code First will apply migrations to database.



Migrate to a Specific Version

There may be times when you want upgrade/downgrade to a specific migration:

- Run the **Update-Database -TargetMigration: %migration-name%** command in Package Manager Console:

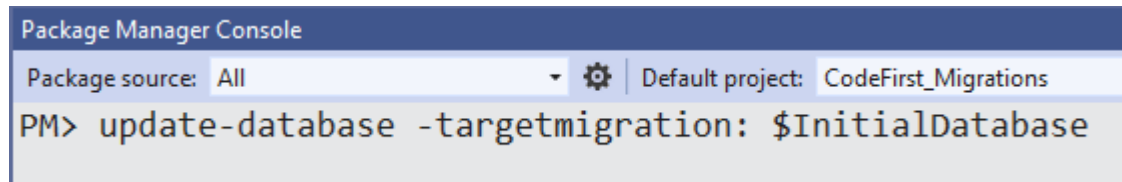


Package Manager Console

Package source: All [gear icon] Default project: CodeFirst_Migrations

```
PM> update-database -targetmigration: AddBlogUrl
```

- If you want to roll all the way back to an empty database then you can run:



Package Manager Console

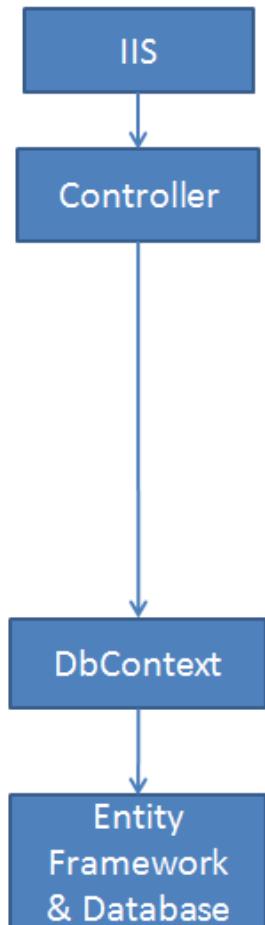
Package source: All [gear icon] Default project: CodeFirst_Migrations

```
PM> update-database -targetmigration: $InitialDatabase
```

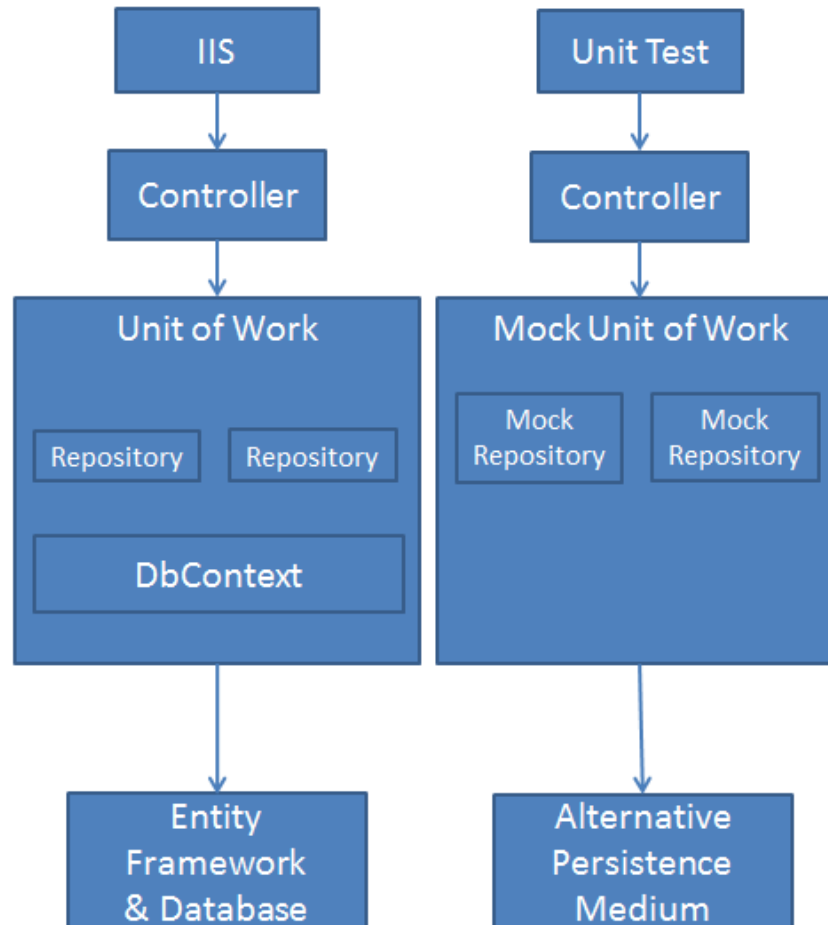

Repository and Unit of Work

Overview

No Repository

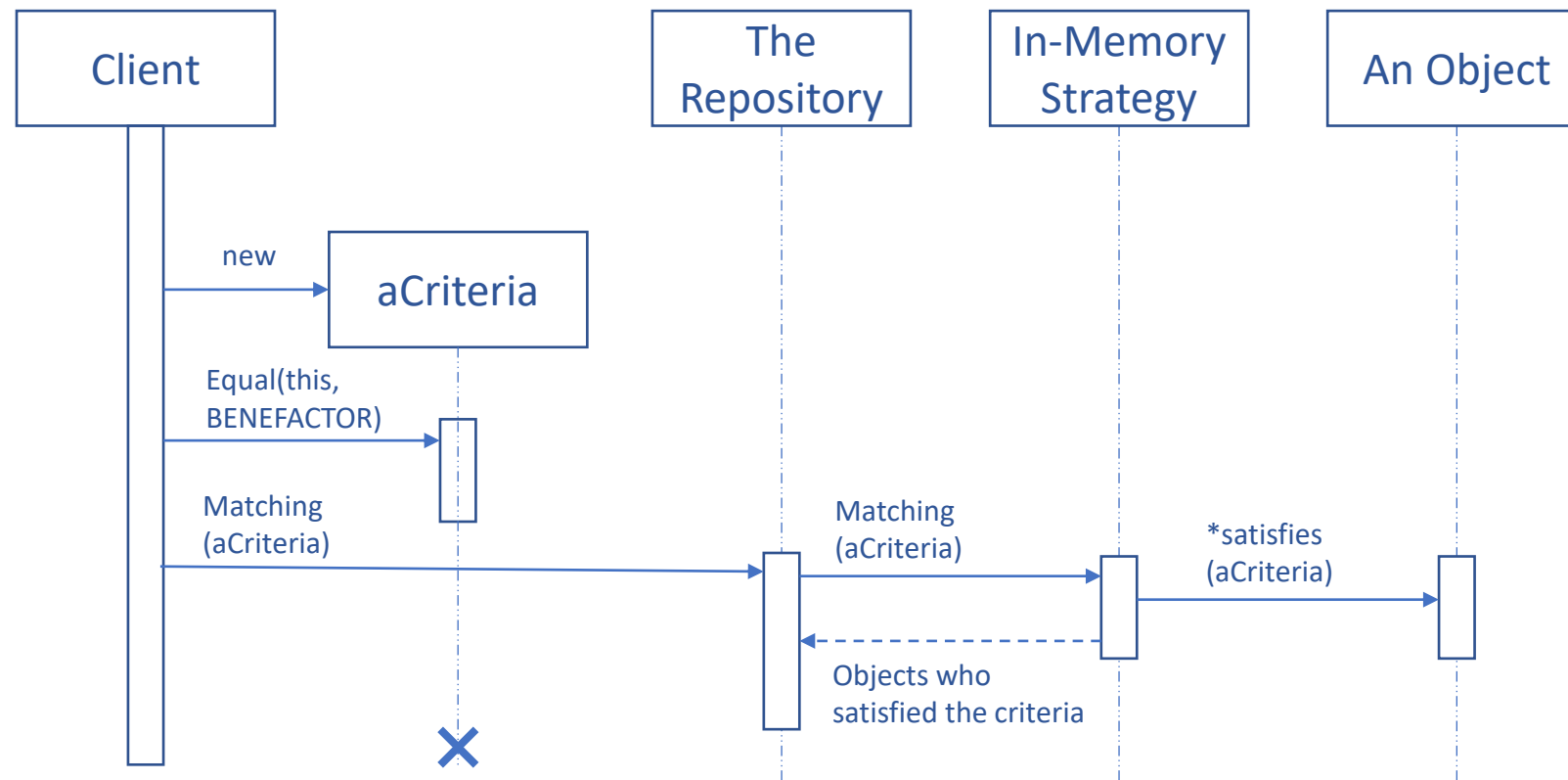


With Repository



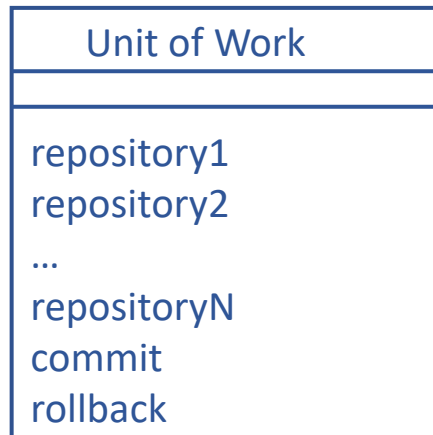
Repository Pattern

Repository - mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects.



Unit of Work Pattern

Unit of Work - maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.



Q&A



DRIVEN



CANDID



CREATIVE



ORIGINAL



INTELLIGENT



EXPERT

UA .NET Online LAB