Overview

- ORM
- Fundamentals
- Creating and configuring a model
- Managing Database Schemas
- Querying Data
- Saving Data
- Database providers

Object Relational Mapping

```
SELECT * FROM blogs WHERE name = 'test';
```

Object-relational-mapping is the idea of being able to write queries like the one above, as well as much more complicated ones, using the object-oriented paradigm of your preferred programming language.

```
using(var context = new BloggingContext()){
    var blogs = context.Blogs.Where(blog=>blog.Name = "test").ToList();
}
```

Pros and cons

1. You get to write in the language you are already using anyway
2. It abstracts away the database system so that switching from MySQL to PostgreSQL, or whatever flavor you prefer, is easy-peasy.
3. Depending on the ORM you get a lot of advanced features out of the box, such as support for transactions, connection pooling, migrations, seeds, streams, and all sorts of other goodies.
4. Many of the queries you write will perform better than if you wrote them yourself.

1. If you are a master at SQL, you can probably get more performant queries by writing them yourself.
2. There is overhead involved in learning how to use any given ORM
3. The initial configuration of an ORM can be a headache.
4. As a developer, it is important to understand what is happening under the hood. Since ORMs can serve as a crutch to avoid understanding databases and SQL, it can make you a weaker developer in that portion of the stack.

Connection string

```
{
  "ConnectionStrings": {
    "BloggingDatabase": "Server=(localdb)\\mssqllocaldb;Database=EFGetStarted.ConsoleApp.NewDb;Trusted_Connection=True;"
  },
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<BloggingContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("BloggingDatabase")));
}
```

Testing

```
[Fact]
public void Add_writes_to_database()
{
    // In-memory database only exists while the connection is open
    var connection = new SqliteConnection("DataSource=:memory:");
    connection.Open();

    try
    {
        var options = new DbContextOptionsBuilder<BloggingContext>()
            .UseSqlite(connection)
            .Options;

        // Create the schema in the database
        using (var context = new BloggingContext(options))
        {
            context.Database.EnsureCreated();
        }

        // Run the test against one instance of the context
        using (var context = new BloggingContext(options))
        {
            var service = new BlogService(context);
            service.Add("http://sample.com");
            context.SaveChanges();
        }

        // Use a separate instance of the context to verify correct data was saved to database
        using (var context = new BloggingContext(options))
        {
            Assert.Equal(1, context.Blogs.Count());
            Assert.Equal("http://sample.com", context.Blogs.Single().Url);
        }
    }
    finally
    {
        connection.Close();
    }
}
```

DbContext

`DbContext` must have an instance of `DbContextOptions` in order to perform any work. The `DbContextOptions` instance carries configuration information such as:

- The database provider to use, typically selected by invoking a method such as `UseSqlServer` or `UseSqlite`. These extension methods require the corresponding provider package, such as `Microsoft.EntityFrameworkCore.SqlServer` or `Microsoft.EntityFrameworkCore.Sqlite`. The methods are defined in the `Microsoft.EntityFrameworkCore` namespace.
- Any necessary connection string or identifier of the database instance, typically passed as an argument to the provider selection method mentioned above
- Any provider-level optional behavior selectors, typically also chained inside the call to the provider selection method
- Any general EF Core behavior selectors, typically chained after or before the provider selector method

Example

```
optionsBuilder
    .UseSqlServer(connectionString, providerOptions=>providerOptions.CommandTimeout(60))
    .UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking);
```

Options

The `DbContextOptions` can be supplied to the `DbContext` by overriding the `OnConfiguring` method or externally via a constructor argument.

If both are used, `OnConfiguring` is applied last and can overwrite options supplied to the constructor argument.

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

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlite("Data Source=blog.db");
    }
}
```

Dependency injection

EF Core supports using `DbContext` with a dependency injection container. Your DbContext type can be added to the service container by using the `AddDbContext<TContext>` method.

`AddDbContext<TContext>` will make both your DbContext type, `TContext`, and the corresponding `DbContextOptions<TContext>` available for injection from the service container.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<BloggingContext>(options => options.UseSqlite("Data Source=blog.db"));
}
```

Arguments

This requires adding a constructor argument to your DbContext type that accepts `DbContextOptions<TContext>`

```
public class BloggingContext : DbContext
{
    public BloggingContext(DbContextOptions<BloggingContext> options)
      :base(options)
    { }

    public DbSet<Blog> Blogs { get; set; }
}
```

Application code

```
public class MyController
{
    private readonly BloggingContext _context;

    public MyController(BloggingContext context)
    {
      _context = context;
    }

    ...
}
```

```
using (var context = serviceProvider.GetService<BloggingContext>())
{
  // do stuff
}

var options = serviceProvider.GetService<DbContextOptions<BloggingContext>>();
```

The Model

With EF Core, data access is performed using a model. A model is made up of entity classes and a context object that represents a session with the database, allowing you to query and save data.

You can generate a model from

- an existing database,
- hand code a model to match your database,
- or use EF Migrations to create a database from your model, and then evolve it as your model changes over time.

Example

```
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;

namespace Intro
{
    public class BloggingContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }
        public DbSet<Post> Posts { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlServer(
                @"Server=(localdb)\mssqllocaldb;Database=Blogging;Integrated Security=True");
        }
    }

    public class Blog
    {
        public int BlogId { get; set; }
        public string Url { get; set; }
        public int Rating { get; set; }
        public List<Post> Posts { get; set; }
    }

    public class Post
    {
        public int PostId { get; set; }
        public string Title { get; set; }
        public string Content { get; set; }

        public int BlogId { get; set; }
        public Blog Blog { get; set; }
    }
}
```

Use fluent API to configure a model

Entity Framework uses a set of conventions to build a model based on the shape of your entity classes. You can specify additional configuration to supplement and/or override what was discovered by convention.

You can override the `OnModelCreating` method in your derived context and use the `ModelBuilder API` to configure your model. This is the most powerful method of configuration and allows configuration to be specified without modifying your entity classes. Fluent API configuration has the highest precedence and will override conventions and data annotations.

```
using Microsoft.EntityFrameworkCore;

namespace EFModeling.FluentAPI.Required
{
    class MyContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Blog>()
                .Property(b => b.Url)
                .IsRequired();
        }
    }

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

Data annotations

```
using Microsoft.EntityFrameworkCore;
using System.ComponentModel.DataAnnotations;

namespace EFModeling.DataAnnotations.Required
{
    class MyContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }
    }

    public class Blog
    {
        public int BlogId { get; set; }
        [Required]
        public string Url { get; set; }
    }
}
```

Including & Excluding Types

Including a type in the model means that EF has metadata about that type and will attempt to read and write instances from/to the database.

Conventions

By convention, types that are exposed in `DbSet` properties on your context are included in your model. In addition, types that are mentioned in the `OnModelCreating` method are also included. Finally, any types that are found by recursively exploring the navigation properties of discovered types are also included in the model.

Example

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<AuditEntry>();
    }
}

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

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog { get; set; }
}

public class AuditEntry
{
    public int AuditEntryId { get; set; }
    public string Username { get; set; }
    public string Action { get; set; }
}
```

Data Annotations

You can use Data Annotations to exclude a type from the model.

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

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

    public BlogMetadata Metadata { get; set; }
}

[NotMapped]
public class BlogMetadata
{
    public DateTime LoadedFromDatabase { get; set; }
}
```

Fluent API

You can use the Fluent API to exclude a type from the model.

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Ignore<BlogMetadata>();
    }
}

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

    public BlogMetadata Metadata { get; set; }
}

public class BlogMetadata
{
    public DateTime LoadedFromDatabase { get; set; }
}
```

Including & Excluding Properties

By convention, public properties with a getter and a setter will be included in the model.

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

    [NotMapped]
    public DateTime
}
```

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Ignore(b => b.LoadedFromDatabase);
    }
}
```

Keys (primary)

A key serves as the primary unique identifier for each entity instance. When using a relational database this maps to the concept of a *primary key*. You can also configure a unique identifier that is not the primary key

By convention, a property named `Id` or `<type name>Id` will be configured as the key of an entity.

```
class Car
{
    public s

    public s
    public s
}
```

```
class MyContext : DbContext
{
    public DbSet<Car> Cars { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Car>()
            .HasKey(c => c.LicensePlate);
    }
}
```

Generated Values

Value Generation Patterns

There are three value generation patterns that can be used for properties:

- No value generation
- Value generated on add
- Value generated on add or update

Generated Values

- No value generation

No value generation means that you will always supply a valid value to be saved to the database. This valid value must be assigned to new entities before they are added to the context.

- Value generated on add

Value generated on add means that a value is generated for new entities.

Depending on the database provider being used, values may be generated client side by EF or in the database. If the value is generated by the database, then EF may assign a temporary value when you add the entity to the context. This temporary value will then be replaced by the database generated value during `SaveChanges()`.

If you add an entity to the context that has a value assigned to the property, then EF will attempt to insert that value rather than generating a new one. A property is considered to have a value assigned if it is not assigned the CLR default value (`null` for `string`, `0` for `int`, `Guid.Empty` for `Guid`, etc.)

Generated Values

- Value generated on add or update

Value generated on add or update means that a new value is generated every time the record is saved (insert or update).

Like `value generated on add`, if you specify a value for the property on a newly added instance of an entity, that value will be inserted rather than a value being generated. It is also possible to set an explicit value when updating. For more information, see Explicit values for generated properties.

By convention, non-composite primary keys of type short, int, long, or Guid will be setup to have values generated on add. All other properties will be setup with no value generation.

Examples

```
public class Blog
{
    [DatabaseGenerated(DatabaseC
    public int BlogId { get; set
    public string Url { get; set
}
```

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
    [DatabaseGenerated(DatabaseGeneratedOption.Computed)]
    public DateTime LastUpdated { get; set; }
}
```

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)
    public DateTime Inserted { get; set; }
}
```

```
modelBuilder.Entity<Blog>()
```

```
modelBuilder.Entity<Blog>()
    .Property(b => b.LastUpdated)
    .ValueGeneratedOnAddOrUpdate();
```

```
    .Property(b => b.Inserted)
    .ValueGeneratedOnAdd();
```

Required and Optional Properties

A property is considered optional if it is valid for it to contain `null`. If `null` is not a valid value to be assigned to a property then it is considered to be a required property.

When mapping to a relational database schema, required properties are created as non-nullable columns, and optional properties are created as nullable columns.

Conventions

By convention, a property whose .NET type can contain null will be configured as optional, whereas properties whose .NET type cannot contain null will be configured as required. For example, all properties with .NET value types (`int`, `decimal`, `bool`, etc.) are configured as required, and all properties with nullable .NET value types (`int?`, `decimal?`, `bool?`, etc.) are configured as optional.

```
public class CustomerWithoutNullableReferenceTypes
{
    public int Id { get; set; }
    [Required]                                  // Data annotations needed to configure as required
    public string FirstName { get; set; }
    [Required]
    public string LastName { get; set; }        // Data annotations needed to configure as required
    public string MiddleName { get; set; }      // Optional by convention
}
```

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

public class Blog
{
    public int BlogId { get; set; }
    [Required]
    public string Url { get; set; }
}
```

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(b => b.Url)
            .IsRequired();
    }
}

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

Max length

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

public class Blog
{
    public int BlogId { get; set; }
    [MaxLength(500)]
    public string Url { get; set; }
}
```

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(b => b.Url)
            .HasMaxLength(500);
    }
}

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

Relationships

A relationship defines how two entities relate to each other. In a relational database, this is represented by a foreign key constraint.

terms used to describe relationships

- Dependent entity: This is the entity that contains the foreign key property(s). Sometimes referred to as the 'child' of the relationship.
- Principal entity: This is the entity that contains the primary/alternate key property(s). Sometimes referred to as the 'parent' of the relationship.
- Foreign key: The property(s) in the dependent entity that is used to store the values of the principal key property that the entity is related to.
- Principal key: The property(s) that uniquely identifies the principal entity. This may be the primary key or an alternate key.
- Navigation property: A property defined on the principal and/or dependent entity that contains a reference(s) to the related entity(s).
    - Collection navigation property: A navigation property that contains references to many related entities.
    - Reference navigation property: A navigation property that holds a reference to a single related entity.
    - Inverse navigation property: When discussing a particular navigation property, this term refers to the navigation property on the other end of the relationship.

The following code listing shows a one-to-many relationship between `Blog` and `Post`

- `Post` is the dependent entity
- `Blog` is the principal entity
- `Post.BlogId` is the foreign key
- `Blog.BlogId` is the principal key (in this case it is a primary key rather than an alternate key)
- `Post.Blog` is a reference navigation property
- `Blog.Posts` is a collection navigation property
- `Post.Blog` is the inverse navigation property of `Blog.Posts` (and vice versa)

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

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

Fully Defined Relationships

By convention, a relationship will be created when there is a navigation property discovered on a type. A property is considered a navigation property if the type it points to can not be mapped as a scalar type by the current database provider.

The most common pattern for relationships is to have navigation properties defined on both ends of the relationship and a foreign key property defined in the dependent entity class.

- If a pair of navigation properties is found between two types, then they will be configured as inverse navigation properties of the same relationship.
- If the dependent entity contains a property named `<primary key property name>`, `<navigation property name><primary key property name>`, or `<principal entity name><primary key property name>`, then it will be configured as the foreign key.

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

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

No Foreign Key Property

While it is recommended to have a foreign key property defined in the dependent entity class, it is not required. If no foreign key property is found, a shadow foreign key property will be introduced with the name `<navigation property name><principal key property name>`

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

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog { get; set; }
}
```

Single Navigation Property

Including just one navigation property (no inverse navigation, and no foreign key property) is enough to have a relationship defined by convention. You can also have a single navigation property and a foreign key property.

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

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
}
```

Cascade Delete

By convention, cascade delete will be set to *Cascade* for required relationships and *ClientSetNull* for optional relationships.

Cascade means dependent entities are also deleted.

ClientSetNull means that dependent entities that are not loaded into memory will remain unchanged and must be manually deleted, or updated to point to a valid principal entity.

For entities that are loaded into memory, EF Core will attempt to set the foreign key properties to null.

Data annotations

There are two data annotations that can be used to configure relationships, `[ForeignKey]` and `[InverseProperty]`. These are available in the `System.ComponentModel.DataAnnotations.Schema` namespace.

[ForeignKey]

You can use the Data Annotations to configure which property should be used as the foreign key property for a given relationship. This is typically done when the foreign key property is not discovered by convention.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }
}

#region Entities
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogForeignKey { get; set; }

    [ForeignKey("BlogForeignKey")]
    public Blog Blog { get; set; }
}
#endregion
```

Data annotations

[InverseProperty]

You can use the Data Annotations to configure how navigation properties on the dependent and principal entities pair up. This is typically done when there is more than one pair of navigation properties between two entity types.

```
class MyContext : DbContext
{
    public DbSet<Post> Posts { get; set; }
    public DbSet<User> Users { get; set; }
}

#region Entities
public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int AuthorUserId { get; set; }
    public User Author { get; set; }

    public int ContributorUserId { get; set; }
    public User Contributor { get; set; }
}

public class User
{
    public string UserId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    [InverseProperty("Author")]
    public List<Post> AuthoredPosts { get; set; }

    [InverseProperty("Contributor")]
    public List<Post> ContributedToPosts { get; set; }
}
#endregion
```

Fluent API

To configure a relationship in the Fluent API, you start by identifying the navigation properties that make up the relationship. `HasOne` or `HasMany` identifies the navigation property on the entity type you are beginning the configuration on. You then chain a call to `WithOne` or `WithMany` to identify the inverse navigation. `HasOne`/`WithOne` are used for reference navigation properties and `HasMany`/`WithMany` are used for collection navigation properties.

```
#region Model
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Post>()
            .HasOne(p => p.Blog)
            .WithMany(b => b.Posts);
    }
}

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

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog { get; set; }
}
#endregion
```

Single Navigation Property

```
#region Model
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .HasMany(b => b.Posts)
            .WithOne();
    }
}

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

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
}
#endregion
```

Foreign Key

```
#region Model
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Post>()
            .HasOne(p => p.Blog)
            .WithMany(b => b.Posts)
            .HasForeignKey(p => p.BlogForeignKey);
    }
}

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

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogForeignKey { get; set; }
    public Blog Blog { get; set; }
}
#endregion
```

Without Navigation Property

```
#region Model
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Post>()
            .HasOne<Blog>()
            .WithMany()
            .HasForeignKey(p => p.BlogId);
    }
}

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

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
}
#endregion
```

Principal Key

```
class MyContext : DbContext
{
    public DbSet<Car> Cars { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<RecordOfSale>()
            .HasOne(s => s.Car)
            .WithMany(c => c.SaleHistory)
            .HasForeignKey(s => s.CarLicensePlate)
            .HasPrincipalKey(c => c.LicensePlate);
    }
}

public class Car
{
    public int CarId { get; set; }
    public string LicensePlate { get; set; }
    public string Make { get; set; }
    public string Model { get; set; }

    public List<RecordOfSale> SaleHistory { get; set; }
}

public class RecordOfSale
{
    public int RecordOfSaleId { get; set; }
    public DateTime DateSold { get; set; }
    public decimal Price { get; set; }

    public string CarLicensePlate { get; set; }
    public Car Car { get; set; }
}
```

One-to-one

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

    public BlogImage BlogImage { get; set; }
}

public class BlogImage
{
    public int BlogImageId { get; set; }
    public byte[] Image { get; set; }
    public string Caption { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

Many-to-many

Many-to-many relationships without an entity class to represent the join table are not yet supported. However, you can represent a many-to-many relationship by including an entity class for the join table and mapping two separate one-to-many relationships.

```
class MyContext : DbContext
{
    public DbSet<Post> Posts { get; set; }
    public DbSet<Tag> Tags { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<PostTag>()
            .HasKey(pt => new { pt.PostId, pt.TagId });

        modelBuilder.Entity<PostTag>()
            .HasOne(pt => pt.Post)
            .WithMany(p => p.PostTags)
            .HasForeignKey(pt => pt.PostId);

        modelBuilder.Entity<PostTag>()
            .HasOne(pt => pt.Tag)
            .WithMany(t => t.PostTags)
            .HasForeignKey(pt => pt.TagId);
    }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public List<PostTag> PostTags { get; set; }
}

public class Tag
{
    public string TagId { get; set; }

    public List<PostTag> PostTags { get; set; }
}
```

Table Mapping

```
[Table("blogs", Schema = "blogging")]
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

Table mapping identifies which table data should be queried from and saved to in the database.

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .ToTable("blogs");
    }
}
```

Column mapping

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

public class Blog
{
    [Column("blog_id")]
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(b => b.BlogId)
            .HasColumnName("blog_id");
    }
}

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

Data types

```
public class Blog
{
    public int BlogId { get; set; }
    [Column(TypeName = "varchar(200)")]
    public string Url { get; set; }
    [Column(TypeName = "decimal(5, 2)")]
    public decimal Rating { get; set; }
}
```

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>(eb =>
        {
            eb.Property(b => b.Url).HasColumnType("varchar(200)");
            eb.Property(b => b.Rating).HasColumnType("decimal(5, 2)");
        });
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
    public decimal Rating { get; set; }
}
```

Default values

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(b => b.Created)
            .HasDefaultValueSql("getdate()");
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
    public DateTime Created { get; set; }
}
```

Managing Database Schemas

EF Core provides two primary ways of keeping your EF Core model and database schema in sync. To choose between the two, decide whether your EF Core model or the database schema is the source of truth.

If you want your EF Core model to be the source of truth, use Migrations. As you make changes to your EF Core model, this approach incrementally applies the corresponding schema changes to your database so that it remains compatible with your EF Core model.

Use Reverse Engineering if you want your database schema to be the source of truth. This approach allows you to scaffold a DbContext and the entity type classes by reverse engineering your database schema into an EF Core model.

Migrations

A data model changes during development and gets out of sync with the database. You can drop the database and let EF create a new one that matches the model, but this procedure results in the loss of data. The migrations feature in EF Core provides a way to incrementally update the database schema to keep it in sync with the application's data model while preserving existing data in the database.

Migrations includes command-line tools and APIs that help with the following tasks:

- Create a migration. Generate code that can update the database to sync it with a set of model changes.
- Update the database. Apply pending migrations to update the database schema.
- Customize migration code. Sometimes the generated code needs to be modified or supplemented.
- Remove a migration. Delete the generated code.
- Revert a migration. Undo the database changes.
- Generate SQL scripts. You might need a script to update a production database or to troubleshoot migration code.
- Apply migrations at runtime. When design-time updates and running scripts aren't the best options, call the `Migrate()` method.

Reverse engineering

Reverse engineering is the process of scaffolding entity type classes and a DbContext class based on a database schema. It can be performed using the `Scaffold-DbContext` command of the EF Core Package Manager Console (PMC) tools or the `dotnet ef dbcontext scaffold` command of the .NET Command-line Interface (CLI) tools.

How it works

Reverse engineering starts by reading the database schema. It reads information about tables, columns, constraints, and indexes.

Next, it uses the schema information to create an EF Core model. Tables are used to create entity types; columns are used to create properties; and foreign keys are used to create relationships.

Finally, the model is used to generate code. The corresponding entity type classes, Fluent API, and data annotations are scaffolded in order to re-create the same model from your app.

Querying Data

Entity Framework Core uses Language Integrated Query (LINQ) to query data from the database. LINQ allows you to use C# (or your .NET language of choice) to write strongly typed queries. It uses your derived context and entity classes to reference database objects. EF Core passes a representation of the LINQ query to the database provider. Database providers in turn translate it to database-specific query language (for example, SQL for a relational database).

Loading all data

C#

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

Loading a single entity

C#

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Single(b => b.BlogId == 1);
}
```

Filtering

C#

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Where(b => b.Url.Contains("dotnet"))
        .ToList();
}
```

Loading Related Data

Entity Framework Core allows you to use the navigation properties in your model to load related entities. There are three common O/RM patterns used to load related data.

- Eager loading means that the related data is loaded from the database as part of the initial query.
- Explicit loading means that the related data is explicitly loaded from the database at a later time.
- Lazy loading means that the related data is transparently loaded from the database when the navigation property is accessed.

Eager loading

You can use the `Include` method to specify related data to be included in query results. In the following example, the blogs that are returned in the results will have their `Posts` property populated with the related posts.

```
using (var contex
{
    var blogs = c
        .Include(
        .ToList()
}
```

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
            .ThenInclude(post => post.Author)
                .ThenInclude(author => author.Photo)
        .Include(blog => blog.Owner)
            .ThenInclude(owner => owner.Photo)
        .ToList();
}
```

```
oggingContext())

gs
og.Posts)
og.Owner)

}
```

Including multiple levels

You can drill down through relationships to include multiple levels of related data using the `ThenInclude` method. The following example loads all blogs, their related posts, and the author of each post.

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
            .ThenInclude(post => post.Author)
        .ToList();
}
```

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
            .ThenInclude(post => post.Author)
                .ThenInclude(author => author.Photo)
        .ToList();
}
```

Explicit loading

You can explicitly load a navigation property via the `DbContext.Entry(...)` API.

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Single(b => b.BlogId == 1);

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

    context.Entry(blog)
        .Reference(b => b.Owner)
        .Load();
}
```

Querying related entities

You can also get a LINQ query that represents the contents of a navigation property.

This allows you to do things such as running an aggregate operator over the related entities without loading them into memory.

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Single(b => b.BlogId == 1);

    var postCount = context.Entry(blog)
        .Collection(b => b.Posts)
        .Query()
        .Count();
}
```

Querying related entities

You can also filter which related entities are loaded into memory.

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Single(b => b.BlogId == 1);

    var goodPosts = context.Entry(blog)
        .Collection(b => b.Posts)
        .Query()
        .Where(p => p.Rating > 3)
        .ToList();
}
```

Lazy loading

The simplest way to use lazy-loading is by installing the Microsoft.EntityFrameworkCore.Proxies package and enabling it with a call to `UseLazyLoadingProxies`

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    => optionsBuilder
        .UseLazyLoadingProxies()
        .UseSqlServer(myConnectionString);
```

```
.AddDbContext<BloggingContext>(
    b => b.UseLazyLoadingProxies()
          .UseSqlServer(myConnectionString));
```

Lazy loading

EF Core will then enable lazy loading for any navigation property that can be overridden--that is, it must be `virtual` and on a class that can be inherited from. For example, in the following entities, the `Post.Blog` and `Blog.Posts` navigation properties will be lazy-loaded.

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

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

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

    public virtual Blog Blog { get; set; }
}
```

Lazy loading without proxies

Lazy-loading proxies work by injecting the `ILazyLoader` service into an entity, as described in Entity Type Constructors

```
public class Blog
{
    private ICollection<Post> _posts;

    public Blog()
    {
    }

    private Blog(ILazyLoader lazyLoader)
    {
        LazyLoader = lazyLoader;
    }

    private ILazyLoader LazyLoader { get; set; }

    public int Id { get; set; }
    public string Name { get; set; }

    public ICollection<Post> Posts
    {
        get => LazyLoader.Load(this, ref _posts);
        set => _posts = value;
    }
}
```

Raw SQL Queries

Entity Framework Core allows you to drop down to raw SQL queries when working with a relational database. Raw SQL queries are useful if the query you want can't be expressed using LINQ. Raw SQL queries are also used if using a LINQ query is resulting in an inefficient SQL query. Raw SQL queries can return regular entity types or keyless entity types that are part of your model.

```
var blogs = context.Blogs
    .FromSqlRaw("SELECT * FROM dbo.Blogs")
    .ToList();
```

```
var user = "johndoe";

var blogs = context.Blogs
    .FromSqlRaw("EXECUTE dbo.GetMostPopularBlogsForUser {0}", user)
    .ToList();
```

Composing with LINQ

```
var searchTerm = ".NET";

var blogs = context.Blogs
    .FromSqlInterpolated($"SELECT * FROM dbo.SearchBlogs({searchTerm})")
    .Where(b => b.Rating > 3)
    .OrderByDescending(b => b.Rating)
    .ToList();
```

```
SELECT [b].[BlogId], [b].[OwnerId], [b].[Rating], [b].[Url]
FROM (
    SELECT * FROM dbo.SearchBlogs(@p0)
) AS [b]
WHERE [b].[Rating] > 3
ORDER BY [b].[Rating] DESC
```

Saving Data

Each context instance has a `ChangeTracker` that is responsible for keeping track of changes that need to be written to the database. As you make changes to instances of your entity classes, these changes are recorded in the `ChangeTracker` and then written to the database when you call `SaveChanges`. The database provider is responsible for translating the changes into database-specific operations (for example, `INSERT`, `UPDATE`, and `DELETE` commands for a relational database).

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.First();
    context.Blogs.Remove(blog);
    context.SaveChanges();
}
```

Providers

Entity Framework Core can access many different databases through plug-in libraries called database providers

NuGet Package	Supported database engines	Maintainer / Vendor	Notes / Requirements	Useful links
Microsoft.EntityFrameworkCore.SqlServer	SQL Server 2012 onwards	EF Core Project (Microsoft)		docs
Microsoft.EntityFrameworkCore.Sqlite	SQLite 3.7 onwards	EF Core Project (Microsoft)		docs
Microsoft.EntityFrameworkCore.InMemory	EF Core in-memory database	EF Core Project (Microsoft)	For testing only	docs
Microsoft.EntityFrameworkCore.Cosmos	Azure Cosmos DB SQL API	EF Core Project (Microsoft)		docs
Npgsql.EntityFrameworkCore.PostgreSQL	PostgreSQL	Npgsql Development Team		docs
Pomelo.EntityFrameworkCore.MySql	MySQL, MariaDB	Pomelo Foundation Project		readme
Pomelo.EntityFrameworkCore.MyCat	MyCAT Server	Pomelo Foundation Project	Prerelease only	readme