# Comparing ADO.NET and Entity Framework Core: A Comprehensive Analysis with Code Examples

July 31, 2025

## Contents

# 1 Introduction

ADO.NET and Entity Framework Core (EF Core) are two prominent data access technologies provided by Microsoft for .NET developers. While both serve the purpose of facilitating interaction with databases, they differ significantly in their approach, abstraction level, and use cases. This document provides a detailed comparison of ADO.NET and EF Core, highlighting their differences, advantages, disadvantages, and practical code examples. The goal is to equip developers with the knowledge to choose the appropriate technology for their projects.

# 2 Overview of ADO.NET

ADO.NET is a low-level data access technology in the .NET Framework, introduced in the early 2000s. It provides a set of classes for connecting to databases, executing queries, and managing data. ADO.NET is highly flexible, allowing developers to work directly with database connections, commands, and data readers.

## 2.1 Key Features of ADO.NET

- **Direct Control**: Offers fine-grained control over database operations.
- **Disconnected Architecture**: Uses DataSets and DataTables for offline data manipulation.
- **Performance**: Optimized for high-performance scenarios due to minimal abstraction.
- **Support for Multiple Databases**: Works with any database that provides an ADO.NET provider.

## 2.2 Advantages of ADO.NET

- High performance for simple queries and bulk operations.
- Full control over SQL queries and database connections.
- Lightweight and suitable for low-level database operations.

## 2.3 Disadvantages of ADO.NET

- Requires manual coding for database operations, increasing development time.
- Prone to errors in SQL query construction.
- Limited support for object-relational mapping (ORM).

# 3 Overview of Entity Framework Core

Entity Framework Core (EF Core) is a modern, lightweight, and cross-platform ORM framework introduced by Microsoft as a successor to Entity Framework. It simplifies

data access by mapping database tables to .NET objects, allowing developers to work with data using C# code instead of raw SQL queries.

### 3.1 Key Features of EF Core

- **ORM Capabilities**: Maps database tables to C# classes automatically.
- **Cross-Platform**: Runs on Windows, Linux, and macOS.
- **LINQ Support**: Enables querying databases using Language Integrated Query (LINQ).
- **Change Tracking**: Automatically tracks changes to objects and persists them to the database.

### 3.2 Advantages of EF Core

- Simplifies data access with ORM, reducing boilerplate code.
- Supports LINQ for type-safe queries.
- Enhances productivity with features like migrations and scaffolding.

### 3.3 Disadvantages of EF Core

- Performance overhead due to abstraction layers.
- Less control over raw SQL queries compared to ADO.NET.
- Steeper learning curve for complex scenarios.

## 4 Key Differences Between ADO.NET and EF Core

The following table summarizes the key differences between ADO.NET and EF Core:

| **Feature** | **ADO.NET** | **EF Core** |
|---|---|---|
| Abstraction Level | Low-level, direct database access | High-level, ORM-based |
| Querying | Raw SQL queries | LINQ and raw SQL |
| Performance | Faster for simple operations | Slower due to ORM overhead |
| Development Speed | Slower, requires manual coding | Faster with automated mappings |
| Cross-Platform | Limited to .NET Framework | Fully cross-platform |
| Change Tracking | Manual | Automatic |

## 5 Code Examples

This section presents code examples demonstrating common database operations using both ADO.NET and EF Core. The examples use a simple database with a `Products` table containing `Id`, `Name`, and `Price` columns.

## 5.1 ADO.NET Code Examples

Below are examples of performing CRUD (Create, Read, Update, Delete) operations using ADO.NET with a SQL Server database.

Listing 1: ADO.NET: Connecting to Database and Reading Data

```
using System;
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString = "Server=localhost;Database=
            SampleDB;Trusted_Connection=True;";
        using (SqlConnection connection = new SqlConnection(
            connectionString))
        {
            connection.Open();
            string sql = "SELECT Id, Name, Price FROM Products";
            using (SqlCommand command = new SqlCommand(sql,
                connection))
            {
                using (SqlDataReader reader = command.
                    ExecuteReader())
                {
                    while (reader.Read())
                    {
                        Console.WriteLine($"ID: {reader["Id"]},
                            Name: {reader["Name"]}, Price: {reader
                            ["Price"]}");
                    }
                }
            }
        }
    }
}
```

Listing 2: ADO.NET: Inserting Data

```
using System;
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString = "Server=localhost;Database=
            SampleDB;Trusted_Connection=True;";
        using (SqlConnection connection = new SqlConnection(
            connectionString))
        {
```

```
            connection.Open();
            string sql = "INSERT INTO Products (Name, Price)
                VALUES (@Name, @Price)";
            using (SqlCommand command = new SqlCommand(sql,
                connection))
            {
                command.Parameters.AddWithValue("@Name", "Laptop"
                    );
                command.Parameters.AddWithValue("@Price", 999.99)
                    ;
                command.ExecuteNonQuery();
            }
        }
    }
}
```

## 5.2 EF Core Code Examples

Below are equivalent CRUD operations using EF Core. Assume a `DbContext` and `Product` entity are defined.

Listing 3: EF Core: DbContext and Entity Definition

```
using Microsoft.EntityFrameworkCore;

public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public double Price { get; set; }
}

public class SampleDbContext : DbContext
{
    public DbSet<Product> Products { get; set; }

    public SampleDbContext(DbContextOptions<SampleDbContext>
        options) : base(options) { }
}
```

Listing 4: EF Core: Reading Data

```
using Microsoft.EntityFrameworkCore;

class Program
{
    static void Main()
    {
        var options = new DbContextOptionsBuilder<SampleDbContext
            >()
            .UseSqlServer("Server=localhost;Database=SampleDB;
                Trusted_Connection=True;")
```

```
            .Options;

        using (var context = new SampleDbContext(options))
        {
            var products = context.Products.ToList();
            foreach (var product in products)
            {
                Console.WriteLine($"ID: {product.Id}, Name: {
                    product.Name}, Price: {product.Price}");
            }
        }
    }
}
```

Listing 5: EF Core: Inserting Data

```
using Microsoft.EntityFrameworkCore;

class Program
{
    static void Main()
    {
        var options = new DbContextOptionsBuilder<SampleDbContext
            >()
            .UseSqlServer("Server=localhost;Database=SampleDB;
                Trusted_Connection=True;")
            .Options;

        using (var context = new SampleDbContext(options))
        {
            var product = new Product { Name = "Laptop", Price =
                999.99 };
            context.Products.Add(product);
            context.SaveChanges();
        }
    }
}
```

# 6 Performance Considerations

ADO.NET typically outperforms EF Core in scenarios involving simple queries or bulk operations due to its lightweight nature. EF Core, however, introduces overhead from its ORM layer, including query translation and change tracking. For complex applications, EF Core's productivity gains often outweigh its performance costs.

# 7 Use Cases

- **ADO.NET**: Best for high-performance applications, legacy systems, or scenarios requiring full control over SQL queries.

- **EF Core**: Ideal for rapid development, cross-platform applications, or projects leveraging LINQ and ORM.

# 8 Conclusion

ADO.NET and EF Core cater to different needs in the .NET ecosystem. ADO.NET offers low-level control and high performance, while EF Core provides a higher abstraction level, improving developer productivity. The choice between them depends on project requirements, performance needs, and development timelines.

# 9 Advanced Features of EF Core

EF Core offers advanced features like lazy loading, eager loading, and explicit loading for managing related data. It also supports database migrations, allowing developers to evolve the database schema over time.

# 10 ADO.NET DataSet vs. EF Core DbContext

The `DataSet` in ADO.NET and `DbContext` in EF Core serve as central components for data manipulation. `DataSet` is a disconnected, in-memory representation of data, while `DbContext` is an ORM-based context for managing entities.

# 11 Security Considerations

ADO.NET requires manual parameterization to prevent SQL injection, while EF Cores LINQ queries are inherently parameterized, reducing the risk of injection attacks.

# 12 Scalability and Maintenance

EF Cores abstraction simplifies maintenance in large applications, while ADO.NETs manual approach may lead to higher maintenance costs but better scalability for specific use cases.

# 13 Future Trends

EF Core continues to evolve with .NET, adding features like improved performance and support for new database providers. ADO.NET remains stable but is less likely to receive significant updates.