Manish Jaiswal

EF Core Series

MASTERING

Entity Framework Core: From Start to Finish

swipe

Manish Jaiswal

EF Core Series

INTRODUCTION TO

Entity Framework Core

swipe

Manish Jaiswal

LINQ Series

01

Overview

Entity Framework Core (EF Core) is a popular **Object-Relational Mapper** (ORM) framework for .NET developers. It simplifies data access by bridging the gap between your application **objects** and **relational databases**. With EF Core, you can focus on your application logic while it handles the complexities of database interactions.

swipe

Manish Jaiswal

LINQ Series

02

History

- Entity Framework (EF) is an open-source Object-Relational Mapper (ORM) for .NET.

- Developed by **Microsoft**, it allows developers to work with a database using .NET objects.

- **Launched in 2008 with .NET Framework 3.5**, EF has evolved significantly over the years.

- EF Core, a **cross-platform** version, was introduced with .NET Core.

swipe

Manish Jaiswal

LINQ Series

03

Key Features and Benefits

- **Code-First development:** Design your domain model using C# classes and let EF Core generate the database schema.

- **Database-First approach:** Reverse engineer your database schema into C# classes for existing databases.

- **Change tracking:** Automatically tracks changes made to your entities and manages updates efficiently.

swipe

Manish Jaiswal

LINQ Series

04

Key Features and Benefits

- **LINQ to Entities:** Query your data using familiar LINQ syntax.
- **Migrations:** Manage database schema changes over time.
- **Cross-platform support:** Works with various databases like SQL Server, SQLite, PostgreSQL, and more.

swipe

Manish Jaiswal

LINQ Series

05

EF Core vs. Other ORMs

While EF Core is a powerful choice, it's essential to consider other ORMs based on your project requirements. Some popular alternatives include:

- **Dapper:** Lightweight micro-ORM focused on performance.
- **NHibernate:** Mature ORM with advanced features.

Choose the ORM that best aligns with your project's needs, considering factors like performance, features, and developer familiarity.

swipe

Manish Jaiswal

LINQ Series

06

Conclusion

- Entity Framework is a robust and versatile ORM that simplifies data access in .NET applications.

- With its rich feature set and continuous improvements, EF remains a top choice for developers.

swipe

Manish Jaiswal

EF Core Series

GETTING

Started with Entity Framework Core

swipe

Manish Jaiswal

EF Core Series

01

Introduction

Entity Framework Core (EF Core) is a **lightweight**, **extensible**, open-source, and cross-platform version of the popular Entity Framework data access technology. It simplifies data access and manipulation in your .NET applications.

swipe

Manish Jaiswal

EF Core Series

02

Installing EF Core 💾

To get started, you need to install the EF Core package. Use the following NuGet command in the Package Manager Console:

```
dotnet add package Microsoft.EntityFrameworkCore
```

swipe

Manish Jaiswal

EF Core Series

03

Setting Up Your First EF Core Project

1. Create a new project:

```
dotnet new console -n EFCoreDemo
cd EFCoreDemo
```

2. Add EF Core packages:

```
dotnet add package Microsoft.EntityFrameworkCore
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

swipe

3. Create a model:

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

Manish Jaiswal

EF Core Series

05

Configuring the DbContext⚙️

Create a DbContext class to manage your database operations:

```
1  public class AppDbContext : DbContext
2  {
3      public DbSet<Product> Products { get; set; }
4
5      protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
6      {
7          optionsBuilder.
8          UseSqlServer(
9            @"Server=(localdb)\mssqllocaldb;Database=EFCoreDemoDb;Trusted_Connection=True;"
10         );
11     }
12 }
```

swipe

Manish Jaiswal

EF Core Series

06

Basic CRUD Operations

Create:

```
1  using (var context = new AppDbContext())
2  {
3      var product = new Product
4      { Name = "Laptop", Price = 999.99M };
5      context.Products.Add(product);
6      context.SaveChanges();
7  }
```

Read:

```
1  using (var context = new AppDbContext())
2  {
3      var product = context.Products.FirstOrDefault(p => p.Name == "Laptop");
4      Console.WriteLine($"Product: {product.Name}, Price: {product.Price}");
5  }
```

swipe

Manish Jaiswal

EF Core Series

07

Updating Records ✏️

```
1   using (var context = new AppDbContext())
2   {
3       var product = context.Products
4       .FirstOrDefault(p => p.Name == "Laptop");
5       if (product != null)
6       {
7           product.Price = 899.99M;
8           context.SaveChanges();
9       }
10  }
```

swipe

Manish Jaiswal

EF Core Series

08

Deleting Records

```
using (var context = new AppDbContext())
{
    var product = context.Products
    .FirstOrDefault(p => p.Name == "Laptop");
    if (product != null)
    {
        context.Products.Remove(product);
        context.SaveChanges();
    }
}
```

swipe

09

Conclusion 🎉

This is a basic introduction. EF Core offers much more, including advanced querying, relationships, migrations, and performance optimizations. Stay tuned for more in-depth content!

Manish Jaiswal

EF Core Series

MASTERING

Entity Framework Code First Approach

swipe

Manish Jaiswal

EF Core Series

01

Introduction

What is Code First?

- An approach where you define your model classes first and then generate the database schema from these classes.
- Provides greater control over the database schema through C# code.

Advantages:

- Aligns with Domain-Driven Design (DDD).
- Simplifies database version control.
- Eases the migration and update process.

swipe

Manish Jaiswal

EF Core Series

02

Creating and Configuring Models

Basic Model Class:

```
1   public class Product
2   {
3       public int ProductId { get; set; }
4       public string Name { get; set; }
5       public decimal Price { get; set; }
6   }
```

swipe

Manish Jaiswal

EF Core Series

03

Data Annotations:

```
1  public class Product
2  {
3      [Key]
4      public int ProductId { get; set; }
5
6      [Required]
7      [MaxLength(100)]
8      public string Name { get; set; }
9
10     [Range(0, 1000)]
11     public decimal Price { get; set; }
12 }
```

Fluent API:

```
1  protected override void OnModelCreating(ModelBuilder modelBuilder)
2  {
3      modelBuilder.Entity<Product>()
4          .Property(p => p.Name)
5          .IsRequired()
6          .HasMaxLength(100);
7  }
```

swipe

Manish Jaiswal

EF Core Series

04

Migrations and Database Updates

Adding a Migration:

```
// Adding a Migration:
// Use the Package Manager Console
Add-Migration InitialCreate

// Or use the .NET CLI
dotnet ef migrations add InitialCreate

```

swipe

05

Applying Migrations:

```
// Use the Package Manager Console
Update-Database

// Or use the .NET CLI
dotnet ef database update

```

Updating the Model:

- **Modify your model classes.**

```
// Add a new migration:
Add-Migration AddNewColumn

// update the database
Update-Database
```

swipe

Manish Jaiswal

EF Core Series

06

Code First in Action

DbContext Class

```
1  public class ApplicationContext : DbContext
2  {
3      public DbSet<Product> Products { get; set; }
4
5      protected override void OnConfiguring
6                (DbContextOptionsBuilder optionsBuilder)
7      {
8          optionsBuilder
9          .UseSqlServer(@"Server=.;Database=CodeFirstDB;Trusted_Connection=True;");
10     }
11
12     protected override void OnModelCreating(ModelBuilder modelBuilder)
13     {
14         modelBuilder.Entity<Product>()
15             .Property(p => p.Name)
16             .IsRequired()
17             .HasMaxLength(100);
18     }
19 }
```

swipe

07

Usage:

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

Manish Jaiswal

EF Core Series

MASTERING

EF Core Database First Approach

swipe

Manish Jaiswal

EF Core Series

01

Introduction

Database First Approach allows you to create a data model based on an existing database. This approach is ideal for projects where the database is already designed and running.

Key Features:

- Works with existing databases
- Auto-generates context and entity classes
- Supports complex database structures

swipe

Manish Jaiswal

EF Core Series

02

Reverse Engineering an Existing Database

To use the Database First Approach, you need to reverse engineer your existing database. Use the Scaffold-DbContext command to generate your context and entity classes.

```
// This command will create a Models directory with
// context and entity classes representing your database.
Scaffold-DbContext "YourConnectionString"
    Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
```

swipe

Manish Jaiswal

EF Core Series

03

Customizing the Generated Models

The generated models can be **customized** to fit your application's needs. You can modify data annotations, relationships, and other properties.

swipe

04

Example

```
1   public partial class Product
2   {
3       [Key]
4       public int ProductId { get; set; }
5
6       [Required]
7       [MaxLength(50)]
8       public string ProductName { get; set; }
9
10      public decimal Price { get; set; }
11  }
```

Here, we've added data annotations to the **Product** model for validation and constraints.

swipe

Manish Jaiswal

EF Core Series

05

Managing Changes with Migrations

Even with Database First, you can manage schema changes using migrations. This allows you to keep your database schema in sync with your model changes.

Example: Adding a Migration

```
1   Add-Migration AddNewColumn
2   Update-Database
```

swipe

Manish Jaiswal

EF Core Series

06

Conclusion and Best Practices

- Regularly update your models by re-scaffolding as the database evolves.
- Keep your models clean and organized by splitting them into partial classes.
- Use migrations to handle schema changes efficiently.

swipe

Manish Jaiswal

EF Core Series

MASTERING

Entity Framework Model First Approach

swipe

Manish Jaiswal

EF Core Series

01

Introduction

Model First is an approach where you create the conceptual model first and then generate the database schema from this model. It is ideal for projects where you prefer to design your database visually.

Benefits:

- Visual modeling of the database schema.
- Automatically generates database from the model.
- It is easy to make changes and update the database.

swipe

Manish Jaiswal

EF Core Series

02

Design Your Model

Use the **Entity Framework Designer** to create your model.

Add a New Model:

- Right-click on your project.
- Select Add > New Item.
- Choose ADO.NET Entity Data Model.
- Select Model First option.

Design the Model:

- Use the designer to add entities and relationships.
- Set properties and configure relationships (one-to-one, one-to-many, many-to-many).

swipe

Manish Jaiswal

EF Core Series

03

Designing a Simple Model

```
// Define an entity for a "Student"
public class Student
{
    public int StudentId { get; set; }
    public string Name { get; set; }
    public DateTime EnrollmentDate { get; set; }
}

// Define an entity for a "Course"
public class Course
{
    public int CourseId { get; set; }
    public string Title { get; set; }
    public int Credits { get; set; }
}
```

In the designer, you would visually add these entities and configure the relationships.

swipe

Manish Jaiswal

EF Core Series

04

Generate Your Database

Once your model is designed, you can generate the database schema.

Generate Database:

- Right-click on the designer surface.
- Select Generate Database from Model.

Configure Connection:

- Set up the connection string to your database.
- Review and execute the generated SQL script.

Update the Database:

- Apply the changes to create or update the database schema.

swipe

Manish Jaiswal

EF Core Series

05

Example: Generated SQL Script

```
1   CREATE TABLE Students (
2     StudentId INT PRIMARY KEY,
3     Name NVARCHAR(MAX),
4     EnrollmentDate DATETIME
5   );
6
7   CREATE TABLE Courses (
8     CourseId INT PRIMARY KEY,
9     Title NVARCHAR(MAX),
10    Credits INT
11  );
12
13  ALTER TABLE Students
14  ADD CONSTRAINT FK_Student_Course
15  FOREIGN KEY (CourseId) REFERENCES Courses(CourseId);
```

swipe

Manish Jaiswal

EF Core Series

06

Code First vs. Model First

Code First:

- Start by writing C# classes.
- The database is generated from the code.
- Ideal for developers who prefer to write code.

Model First:

- Start by designing the model visually.
- The database is generated from the model.
- Ideal for visualizing and managing the database schema.

swipe

Manish Jaiswal

EF Core Series

MASTERING

Entity Framework Core Performance Optimization

swipe

Manish Jaiswal

EF Core Series

01

Introduction

Entity Framework Core (EF Core) is powerful, but performance can be a concern. Let's explore best practices, query optimization techniques, and caching strategies to enhance performance.

swipe

Manish Jaiswal

EF Core Series

02

Best Practices for Performance

01. Tracking Changes: Track only the entities you need to modify using AsTracking or AsNoTracking. Unnecessary tracking overhead can impact performance.

```
1   var customerToUpdate = _context.Customers.AsTracking()
2                                 .FirstOrDefault(c => c.Id == 1);
```

swipe

02. Lazy Loading vs. Eager Loading:

Utilize lazy loading for relationships accessed infrequently. Eager loading is efficient for frequently accessed data, reducing round trips to the database.

```
// Lazy Loading
var customer = _context.Customers.Find(1);
// Orders loaded only when accessed
customer.Orders.ToList();

// Eager Loading
var customer = _context.Customers.Include(c => c.Orders)
                .FirstOrDefault(c => c.Id == 1);
```

03. Batch Save Changes:

```
context.BulkInsert(customers);
context.BulkSaveChanges();
```

swipe

Manish Jaiswal

EF Core Series

04

Query Optimization Techniques

```
// 1) Select Specific Fields:
var userNames = context.Customers
                .Select(c => new { c.Name }).ToList();


// 2) Use Compiled Queries:
var compiledQuery = EF.CompileQuery((MyDbContext ctx) =>
  ctx.Customers.Where(u => u.Age > 18));
var customers = compiledQuery(context).ToList();


// 3) Filter early
var activeCustomers = context.Customers
                      .Where(u => u.IsActive).ToList();
```

swipe

Manish Jaiswal

EF Core Series

05

Caching Strategies

01. In-Memory Caching: Cache frequently accessed data in-memory using libraries like MemoryCache for faster retrieval.

```
var activeCustomers = context.Customers
                      .Where(u => u.IsActive).ToList();


var cache = new MemoryCache();
var cacheKey = $"customer-{customerId}";
if (!cache.TryGetValue(cacheKey, out Customer customer))
{
  customer = _context.Customers.Find(customerId);
  cache.Set(cacheKey, customer, TimeSpan.FromMinutes(30));
}
```

swipe

Manish Jaiswal

EF Core Series

02. Response Caching

```
1  [ResponseCache(Duration = 60)]
2  public IActionResult GetUsers()
3  {
4      var users = context.Users.ToList();
5      return Ok(users);
6  }
```

03. Distributed Cache:

```
1  services.AddStackExchangeRedisCache(options =>
2  {
3      options.Configuration = "localhost:6379";
4  });
```

swipe

Manish Jaiswal

EF Core Series

MASTERING

EF Core Migrations

swipe

Manish Jaiswal

EF Core Series

01

Understanding Migrations

Entity Framework Core Migrations is a powerful feature that simplifies database **schema management** alongside your application's evolution. It automates the process of creating and applying changes to your database schema, ensuring your database remains in sync with your evolving data models.

swipe

Manish Jaiswal

EF Core Series

02

Creating and Applying Migrations

```
1  // Step 1: Add a new migration
2  dotnet ef migrations add InitialCreate
3
4  // Step 2: Apply the migration to the database
5  dotnet ef database update
```

- The **add command** creates a migration file with the necessary schema changes.
- The **update command** applies these changes to the database.

swipe

Manish Jaiswal

EF Core Series

03

Handling Schema Changes

Adding a Column:

```
1  public class MyEntity
2  {
3      public int Id { get; set; }
4      public string NewColumn { get; set; } // New Column
5  }
```

```
dotnet ef migrations add AddNewColumn
dotnet ef database update
```

swipe

Manish Jaiswal

EF Core Series

04

Removing a Column

```
1   public class MyEntity
2   {
3       public int Id { get; set; }
4       // Remove the column
5   }
```

```
dotnet ef migrations add RemoveColumn
dotnet ef database update
```

swipe

Manish Jaiswal

EF Core Series

05

Best Practices for Migrations

- Keep migrations small and manageable.
- Test migrations in a staging environment before applying to production.
- Use meaningful names for migration files.

swipe

Manish Jaiswal

EF Core Series

06

Tips for Smooth Migration Process

- Always backup your database before applying migrations.
- Review the generated SQL script to understand the changes being applied.
- Keep the migration history clean by removing obsolete migrations if possible.

swipe

Manish Jaiswal

EF Core Series

MASTERING

Entity Framework Relationships

swipe

Manish Jaiswal

EF Core Series

01

Introduction to EF Core Relationships

Entity Framework Core (EF Core) empowers you to model complex data relationships within your applications. Understanding these relationships is fundamental to building robust and efficient data models. The core concepts of EF Core relationships include:

- **One-to-One**
- **One-to-Many**
- **Many-to-Many**

swipe

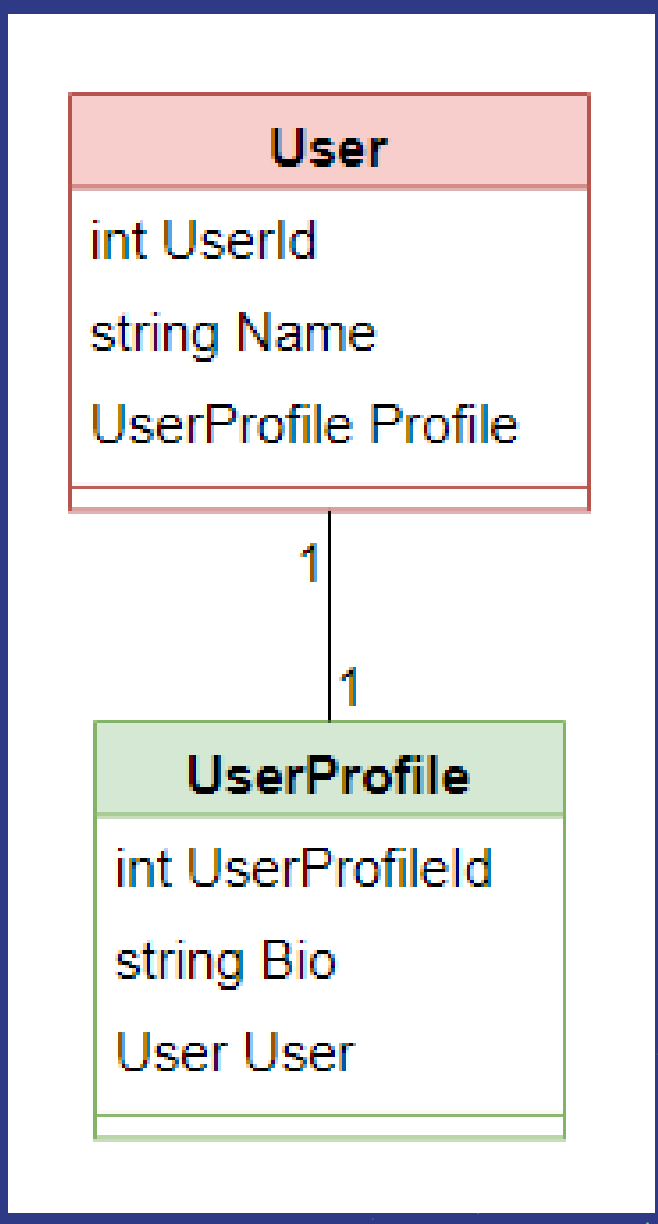
Manish Jaiswal

EF Core Series

02

One-to-One Relationships

In a One-to-One relationship, each entity instance is associated with a single instance of another entity.



swipe

Manish Jaiswal

EF Core Series

03

```
1   public class User
2   {
3       public int UserId { get; set; }
4       public string Name { get; set; }
5       public UserProfile Profile { get; set; }
6   }
7
8   public class UserProfile
9   {
10      public int UserProfileId { get; set; }
11      public string Bio { get; set; }
12      public User User { get; set; }
13  }
14
15  modelBuilder.Entity<User>()
16      .HasOne(u => u.Profile)
17      .WithOne(p => p.User)
18      .HasForeignKey<UserProfile>(p => p.UserProfileId);
```

swipe

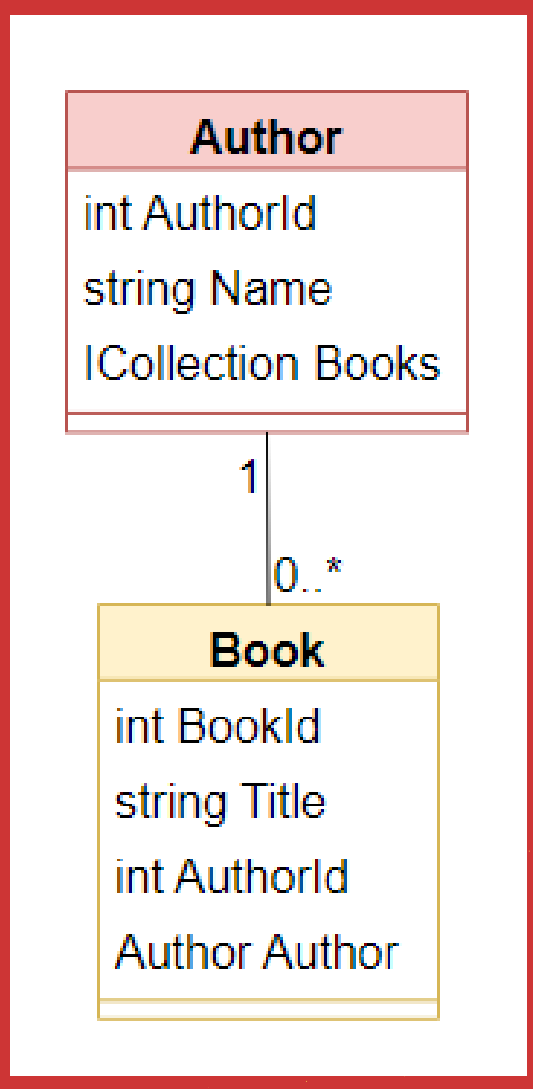
Manish Jaiswal

EF Core Series

04

One-to-Many Relationships

In a One-to-Many relationship, a single entity instance is associated with multiple instances of another entity.



swipe

Manish Jaiswal

EF Core Series

05

```
public class Author
{
    public int AuthorId { get; set; }
    public string Name { get; set; }
    public ICollection<Book> Books { get; set; }
}

public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public int AuthorId { get; set; }
    public Author Author { get; set; }
}

modelBuilder.Entity<Author>()
    .HasMany(a => a.Books)
    .WithOne(b => b.Author)
    .HasForeignKey(b => b.AuthorId);
```

swipe

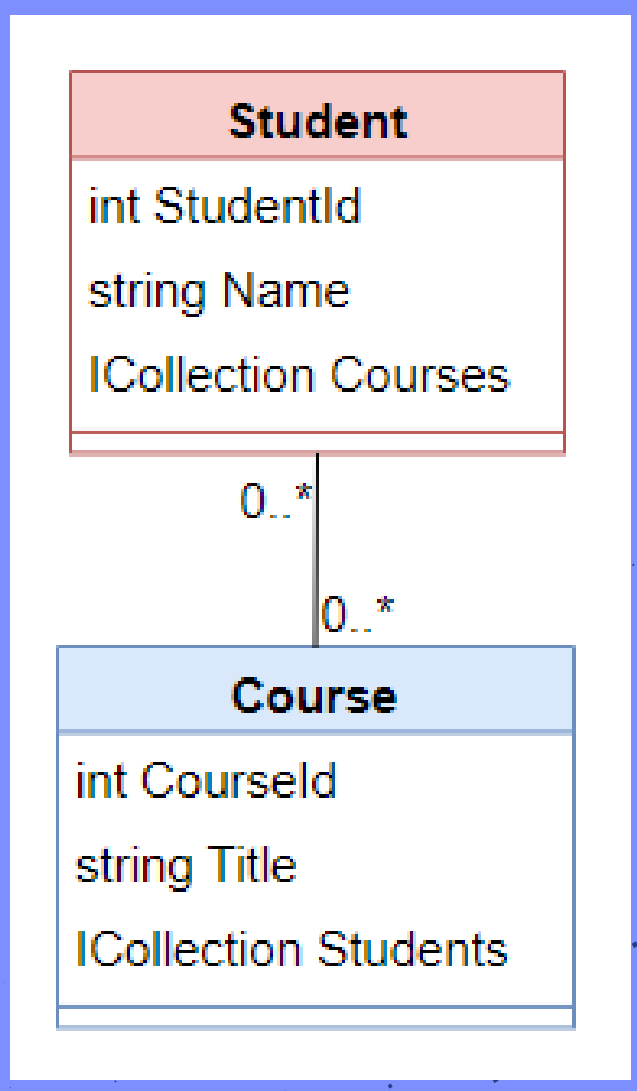
Manish Jaiswal

EF Core Series

06

Many-to-Many Relationships

In a Many-to-Many relationship, multiple instances of one entity are associated with multiple instances of another entity.



swipe

Manish Jaiswal

EF Core Series

07

```
public class Student
{
    public int StudentId { get; set; }
    public string Name { get; set; }
    public ICollection<Course> Courses { get; set; }
}

public class Course
{
    public int CourseId { get; set; }
    public string Title { get; set; }
    public ICollection<Student> Students { get; set; }
}

modelBuilder.Entity<Student>()
    .HasMany(s => s.Courses)
    .WithMany(c => c.Students)
    .UsingEntity(j => j.ToTable("StudentCourses"));
```

swipe

Manish Jaiswal

EF Core Series

08

Navigational Properties

Navigational properties are properties on an entity that hold references to related entities, allowing you to navigate from one entity to its related entities.

Example: In the previous examples, the Profile property in the User class, the Books collection in the Author class, and the Courses collection in the Student class are navigational properties.

swipe

Manish Jaiswal

EF Core Series

09

Cascade Delete

Cascade delete automatically deletes dependent entities when the principal entity is deleted. This helps maintain data integrity by ensuring no orphaned records.

```
1   modelBuilder.Entity<Author>()
2   .HasMany(a => a.Books)
3   .WithOne(b => b.Author)
4   .OnDelete(DeleteBehavior.Cascade);
```

swipe

Manish Jaiswal

EF Core Series

10

Handling Orphaned Records

Orphaned records occur when a principal entity is deleted, but its related entities are not. Using cascade delete can prevent this. Alternatively, you can configure **DeleteBehavior.Restrict** or **DeleteBehavior.SetNull**.

```
1   modelBuilder.Entity<Author>()
2       .HasMany(a => a.Books)
3       .WithOne(b => b.Author)
4       .OnDelete(DeleteBehavior.Restrict);
5       // Or DeleteBehavior.SetNull
```

swipe

Manish Jaiswal

EF Core Series

11

Conclusion and Best Practices

- Always define relationships explicitly in your model.
- Use navigational properties for easy data access.
- Implement cascade delete carefully to avoid unintended data loss.
- Regularly review and update your data model as your application evolves.

swipe

Manish Jaiswal

EF Core Series

MASTERING

EF Core: Data Annotations vs. Fluent API

swipe

Manish Jaiswal

EF Core Series

01

Understanding Data Annotations

Data Annotations are attributes you can apply to your model classes and properties to configure the behavior of EF Core. These attributes provide a simple way to specify **metadata** about the models.

swipe

Manish Jaiswal

EF Core Series

02

Example

```
1   public class Student
2   {
3       [Key]
4       public int StudentId { get; set; }
5
6       [Required]
7       [MaxLength(50)]
8       public string Name { get; set; }
9
10      [Range(1, 100)]
11      public int Age { get; set; }
12  }
```

swipe

Manish Jaiswal

EF Core Series

03

Key Data Annotations to Know

Here are some commonly used Data Annotations:

- [**Key**] - Specifies the primary key.
- [**Required**] - Marks a property as required.
- [**MaxLength**] - Sets the maximum length of a string field.
- [**Range**] - Defines a numeric range.

swipe

Manish Jaiswal

EF Core Series

04

Example

```
1   public class Course
2   {
3       [Key]
4       public int CourseId { get; set; }
5
6       [Required]
7       [MaxLength(100)]
8       public string Title { get; set; }
9
10      [Range(1, 10)]
11      public int Credits { get; set; }
12  }
```

swipe

Manish Jaiswal

EF Core Series

05

Getting Started with Fluent API

Fluent API provides more advanced configuration options compared to Data Annotations. It is defined in the **OnModelCreating** method of your **DbContext** class.

swipe

Manish Jaiswal

EF Core Series

06

Example

```
public class SchoolContext : DbContext
{
    public DbSet<Student> Students { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Student>()
            .HasKey(s => s.StudentId);

        modelBuilder.Entity<Student>()
            .Property(s => s.Name)
            .IsRequired()
            .HasMaxLength(50);

        modelBuilder.Entity<Student>()
            .Property(s => s.Age)
            .HasRange(1, 100);
    }
}
```

swipe

Manish Jaiswal

EF Core Series

07

Advanced Fluent API Configurations

Fluent API offers extensive configuration options, such as:

- **HasKey** - Defines the primary key.
- **Property** - Configures properties.
- **HasMaxLength** - Sets the maximum length.
- **IsRequired** - Specifies required fields.

swipe

Manish Jaiswal

EF Core Series

08

Example

```
modelBuilder.Entity<Course>()
    .HasKey(c => c.CourseId);

modelBuilder.Entity<Course>()
    .Property(c => c.Title)
    .IsRequired()
    .HasMaxLength(100);

modelBuilder.Entity<Course>()
    .Property(c => c.Credits)
    .HasRange(1, 10);
```

swipe

Manish Jaiswal

EF Core Series

09

Data Annotations vs. Fluent API

Data Annotations:

- Simpler to use.
- Ideal for straightforward configurations.
- Limited in flexibility.

Fluent API:

- Offers advanced configurations.
- Provides more control over model behavior.
- Recommended for complex scenarios.

swipe

Manish Jaiswal

EF Core Series

10

When to Use Which?

- Use Data Annotations for simple, quick configurations.
- Opt for Fluent API for complex, extensive configurations.
- Both methods can be used together, but Fluent API **overrides** Data Annotations.

swipe

Manish Jaiswal

EF Core Series

11

Key Takeaways

- Data Annotations are great for simplicity.
- Fluent API provides advanced configuration options.
- Choose based on the complexity of your model requirements.

swipe

Manish Jaiswal

EF Core Series

MASTERING

Concurrency Control in EF Core

swipe

Manish Jaiswal

EF Core Series

01

Concurrency Control Basics

Concurrency control ensures data consistency when multiple users access and modify data concurrently. Entity Framework provides built-in mechanisms to handle **concurrency conflicts** effectively.

swipe

Manish Jaiswal

EF Core Series

02

Dealing with Conflicts

When multiple users attempt to update the same data, conflicts arise. Entity Framework uses concurrency tokens to detect and resolve these conflicts.

```
1   public class Product
2   {
3       public int ProductId { get; set; }
4       public string Name { get; set; }
5       public decimal Price { get; set; }
6       [Timestamp]
7       public byte[] RowVersion { get; set; }
8   }
```

swipe

Manish Jaiswal

EF Core Series

03

Optimistic Concurrency Control

Optimistic concurrency assumes conflicts are rare and checks for conflicts before committing changes. Use the [**Timestamp**] attribute or a custom property to track changes.

swipe

Manish Jaiswal

EF Core Series

04

```
try
{
    context.SaveChanges();
}
catch (DbUpdateConcurrencyException ex)
{
    // Handle concurrency conflict
    var entry = ex.Entries.Single();
    var clientValues = (Product)entry.Entity;
    var databaseEntry = entry.GetDatabaseValues();
    var databaseValues = (Product)databaseEntry.ToObject();

    // Update client values with database values or handle as needed
}
```

swipe

Manish Jaiswal

EF Core Series

05

Pessimistic Concurrency

Pessimistic concurrency locks the data to prevent other users from modifying it until the current transaction is complete. This approach is less common in Entity Framework but can be implemented with explicit locks.

```
using (var transaction = context.Database
    .BeginTransaction(System.Data.IsolationLevel.Serializable))
{
    // Perform data operations
    context.SaveChanges();
    transaction.Commit();
}
```

swipe

Manish Jaiswal

EF Core Series

06

Implementing Custom Control

Custom concurrency control can be achieved by manually handling conflict detection and resolution logic tailored to your application's needs.

swipe

Manish Jaiswal

EF Core Series

07

```
1   public async Task UpdateProductAsync(Product updatedProduct)
2   {
3       var product = await context.Products.FindAsync(updatedProduct.ProductId);
4       if (product.RowVersion.SequenceEqual(updatedProduct.RowVersion))
5       {
6           context.Entry(product).CurrentValues.SetValues(updatedProduct);
7           await context.SaveChangesAsync();
8       }
9       else
10      {
11          // Handle conflict
12          throw new DbUpdateConcurrencyException();
13      }
14  }
```

swipe

Manish Jaiswal

EF Core Series

08

Ensuring Data Integrity

Effective concurrency control in Entity Framework ensures data integrity and a smooth user experience. Choose the right strategy based on your application's requirements.

swipe

Manish Jaiswal

EF Core Series

MASTERING

Entity Framework Transactions

swipe

Manish Jaiswal

EF Core Series

01

Understanding Transactions

- Transactions ensure data integrity by treating **multiple operations** as a single unit.
- They are essential for maintaining **consistency**, especially in complex applications.

swipe

Manish Jaiswal

EF Core Series

02

Built-in Transaction Management

- Entity Framework (EF) provides built-in transaction management through the **DbContext**.
- EF automatically handles transactions for single **SaveChanges** calls.

```
using (var context = new AppDbContext())
{
    // Operations
    context.SaveChanges(); // EF handles transaction automatically
}
```

swipe

Manish Jaiswal

EF Core Series

03

Handling Multiple Operations as a Unit

Transactions are crucial for ensuring data **integrity** when performing multiple related operations. For instance, transferring money between accounts involves debiting one account and crediting another. Both operations should be part of a single transaction:

swipe

Manish Jaiswal

EF Core Series

04

Example

```
using (var transaction = context.Database.BeginTransaction())
{
    try
    {
        // Debit one account
        // Credit another account
        context.SaveChanges();
        transaction.Commit();
    }
    catch (Exception ex)
    {
        transaction.Rollback();
        // Handle the exception
    }
}
```

swipe

Manish Jaiswal

EF Core Series

05

Rollback Strategies

- Rollbacks revert all changes if any operation fails.
- Essential for maintaining **data integrity** in case of errors.

```
using (var context = new AppDbContext())
{
    using (var transaction = context.Database.BeginTransaction())
    {
        try
        {
            // Operations
            transaction.Commit();
        }
        catch (Exception)
        {
            transaction.Rollback();
        }
    }
}
```

swipe

Manish Jaiswal

EF Core Series

06

Commit Strategies

- Use **Commit** to finalize transactions after successful operations.
- Always wrap Commit within a **try-catch** block to handle exceptions.

```
1  using (var context = new AppDbContext())
2  {
3      using (var transaction = context.Database.BeginTransaction())
4      {
5          try
6          {
7              // Operations
8              transaction.Commit();
9          }
10         catch (Exception)
11         {
12             transaction.Rollback();
13             // Log or handle the exception
14         }
15     }
16 }
```

swipe

Manish Jaiswal

EF Core Series

07

Combining Explicit Transactions with Multiple Contexts

- Handle transactions across **multiple DbContext** instances.
- Ensure consistency across different data sources.

swipe

Manish Jaiswal

EF Core Series

08

```
using (var context1 = new AppDbContext1())
using (var context2 = new AppDbContext2())
{
    using (var transaction = context1.Database.BeginTransaction())
    {
        try
        {
            // Operations on context1
            context1.SaveChanges();

            // Operations on context2
            context2.Database
                    .UseTransaction(transaction.GetDbTransaction());
            context2.SaveChanges();

            transaction.Commit();
        }
        catch (Exception)
        {
            transaction.Rollback();
        }
    }
}
```

swipe

Manish Jaiswal

EF Core Series

09

Key Takeaways

- Transactions are critical for data integrity.
- EF provides built-in and explicit transaction management.
- Rollbacks and commits ensure consistency and handle errors effectively.

swipe

Manish Jaiswal

EF Core Series

MASTERING

Entity Framework Change Tracking

swipe

Manish Jaiswal

EF Core Series

01

Understanding Change Tracking

Change tracking is a mechanism that allows EF Core to keep track of changes to your entities. This helps EF Core determine what SQL commands need to be executed to persist changes to the database.

swipe

Manish Jaiswal

EF Core Series

02

Example

```
using (var context = new AppDbContext())
{
    var blog = context.Blogs
                .FirstOrDefault(b => b.BlogId == 1);
    blog.Title = "Updated Title";
    // EF Core tracks the change in the Title property
}

```

swipe

Manish Jaiswal

EF Core Series

03

Managing State of Entities

EF Core tracks the state of each entity (Added, Unchanged, Modified, Deleted) to determine the operations to be performed during **SaveChanges**.

```
using (var context = new AppDbContext())
{
    var blog = context.Blogs
                .FirstOrDefault(b => b.BlogId == 1);
    context.Entry(blog).State = EntityState.Modified;
    context.SaveChanges();
    // The state is set to Modified, prompting an UPDATE command
}
```

swipe

Manish Jaiswal

EF Core Series

04

Using No-Tracking Queries

No-tracking queries improve performance for **read-only operations** by not tracking the entities in the context. This reduces memory usage and speeds up query execution.

```
using (var context = new AppDbContext())
{
    var blogs = context.Blogs.AsNoTracking().ToList();
    // No change tracking is performed, ideal for read-only operations
}
```

swipe

Manish Jaiswal

EF Core Series

05

Benefits of No-Tracking Queries

- Performance: Faster queries and lower memory footprint.
- Scalability: Better for large-scale read operations.
- Simplicity: Reduces overhead when changes to entities are not needed.

```
using (var context = new AppDbContext())
{
    var post = context.Posts.AsNoTracking().FirstOrDefault(p => p.PostId == 1);
    // Efficiently retrieves the post without tracking its state
}
```

swipe

06

Summary

- **Change Tracking:** Tracks changes to determine necessary database operations.
- **Entity State Management:** Manages entity states for accurate database updates.
- **No-Tracking Queries:** Optimizes performance for read-only operations.

Manish Jaiswal

EF Core Series

MASTERING

Entity Framework Core with Dependency Injection

swipe

Manish Jaiswal

EF Core Series

01

Understanding the Basics

Entity Framework Core (EF Core) is an **Object-Relational Mapper** (ORM) that simplifies data access in .NET applications. **Dependency Injection** (DI) is a design pattern that promotes loose coupling and testability. Combining these two powerful tools can significantly enhance your application's architecture.

swipe

Manish Jaiswal

EF Core Series

02

Integrating EF Core with DI

To integrate EF Core with DI, you typically register the DbContext in your DI container. This allows you to inject the **DbContext** into your services and repositories, promoting better testability and maintainability.

```
1  public void ConfigureServices(IServiceCollection services)
2  {
3      services.AddDbContext<MyDbContext>(options =>
4          options.UseSqlServer(connectionString));
5  }
```

swipe

Manish Jaiswal

EF Core Series

03

Configuring DbContext in DI Container with Lifetime

Properly configuring the DbContext lifetime is crucial for optimal performance and resource management. Consider the following options:

Scoped: Create a new DbContext instance for each request, suitable for most web applications.

swipe

Manish Jaiswal

EF Core Series

04

Singleton: Create a single DbContext instance for the entire application lifetime, suitable for specific scenarios.

Transient: Create a new DbContext instance for each dependency injection resolution, useful for testing.

```
services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")),
    ServiceLifetime.Transient);
```

swipe

Manish Jaiswal

EF Core Series

05

Benefits of Integrating EF with DI

- Separation of Concerns: Keeps data access logic separate from business logic.
- Testability: Easier to mock DbContext for unit testing.
- Maintainability: Centralized configuration of DbContext.

swipe

Manish Jaiswal

EF Core Series

MASTERING

Testing with Entity Framework Core

swipe

Manish Jaiswal

EF Core Series

01

Why Testing is Crucial ?

Testing ensures your Entity Framework Core applications are reliable, maintainable, and free of bugs. Proper testing strategies can save time and resources in the long run.

swipe

Manish Jaiswal

EF Core Series

02

Unit Testing EF Core Applications

Unit testing focuses on testing **individual components** in isolation. For EF Core, it means testing your repository or service logic without hitting the actual database.

swipe

Manish Jaiswal

EF Core Series

03

Example

```
1  public class ProductServiceTests
2  {
3      private readonly ProductService _service;
4      private readonly Mock<IProductRepository> _mockRepo;
5
6      public ProductServiceTests()
7      {
8          _mockRepo = new Mock<IProductRepository>();
9          _service = new ProductService(_mockRepo.Object);
10     }
11
12     [Fact]
13     public void GetProduct_ShouldReturnProduct_WhenProductExists()
14     {
15         var productId = 1;
16         _mockRepo.Setup(repo => repo.GetProduct(productId))
17                  .Returns(new Product
18                      { Id = productId,
19                        Name = "Test Product"
20                      });
21
22         var result = _service.GetProduct(productId);
23
24         Assert.NotNull(result);
25         Assert.Equal("Test Product", result.Name);
26     }
27 }
```

swipe

Manish Jaiswal

EF Core Series

04

Mocking DbContext for Tests

Mocking **DbContext** helps you simulate database interactions in your tests, ensuring your business logic is tested without relying on a real database.

swipe

Manish Jaiswal

EF Core Series

05

Example

```

public class DbContextMock
{
    public static Mock<DbSet<T>> GetDbSet<T>(List<T> entities) where T : class
    {
        var queryable = entities.AsQueryable();
        var dbSet = new Mock<DbSet<T>>();

        dbSet.As<IQueryable<T>>()
        .Setup(m => m.Provider).Returns(queryable.Provider);

        dbSet.As<IQueryable<T>>()
        .Setup(m => m.Expression).Returns(queryable.Expression);

        dbSet.As<IQueryable<T>>()
        .Setup(m => m.ElementType).Returns(queryable.ElementType);

        dbSet.As<IQueryable<T>>()
        .Setup(m => m.GetEnumerator()).Returns(queryable.GetEnumerator());
        return dbSet;
    }
}
```

swipe

Manish Jaiswal

EF Core Series

06

Integration Testing with In-Memory Databases

Integration tests verify that different parts of the application work together. Using in-memory databases like InMemoryDatabase ensures your tests run fast and in isolation.

swipe

Manish Jaiswal

EF Core Series

```
public class ProductServiceIntegrationTests
{
    private readonly DbContextOptions<AppDbContext> _options;

    public ProductServiceIntegrationTests()
    {
        _options = new DbContextOptionsBuilder<AppDbContext>()
                    .UseInMemoryDatabase(databaseName: "TestDatabase")
                    .Options;
    }

    [Fact]
    public void GetProduct_ShouldReturnProduct_WhenProductExists()
    {
        using (var context = new AppDbContext(_options))
        {
            context.Products.Add(new Product
                                { Id = 1,
                                  Name = "Integration Test Product"
                                });
            context.SaveChanges();
        }

        using (var context = new AppDbContext(_options))
        {
            var service = new ProductService(context);
            var result = service.GetProduct(1);

            Assert.NotNull(result);
            Assert.Equal("Integration Test Product", result.Name);
        }
    }
}
```

swipe

Manish Jaiswal

EF Core Series

08

Best Practices for Testing with EF Core

- Use in-memory databases for integration tests.
- Mock DbContext for unit tests.
- Isolate tests to ensure reliability.
- Always clean up data after tests.

swipe

Manish Jaiswal

EF Core Series

MASTERING

Entity Framework in Microservices Architecture

swipe

Manish Jaiswal

EF Core Series

01

Introduction

Entity Framework Core (EF Core) is a powerful ORM for .NET applications. But how does it fit into the microservices world?

- **Decoupling:** Each microservice has its own database and DbContext, promoting independence.
- **Data Ownership**: Clear ownership of data within each service.
- **Scalability:** Independent scaling of microservices and databases.

swipe

Manish Jaiswal

EF Core Series

02

Using EF Core in Microservices

Let's dive into the practical aspects of using EF Core in microservices.

- **Dedicated DbContext:** Create a separate DbContext for each microservice.
- **Dependency Injection:** Use DI to manage DbContext instances.
- **Migrations:** Manage database schema changes independently for each service.

swipe

Manish Jaiswal

EF Core Series

03

Example

```
1  public class OrderServiceDbContext : DbContext
2  {
3      public DbSet<Order> Orders { get; set; }
4
5      public OrderServiceDbContext(
6          DbContextOptions<OrderServiceDbContext> options)
7          : base(options)
8      {
9      }
10 }
```

swipe

Manish Jaiswal

EF Core Series

04

Sharing Models and DbContext

While sharing models might seem tempting, it's generally not recommended.

- **Tight Coupling:** Shared models can lead to tight coupling between services.
- **Data Consistency Challenges:** Maintaining data consistency across services becomes difficult.
- **Alternative Approaches:** Consider using DTOs or CQRS for data transfer.

swipe

Manish Jaiswal

EF Core Series

05

Handling Distributed Transactions

In a microservices architecture, handling transactions that span multiple services can be challenging. EF Core provides mechanisms to handle distributed transactions using tools like the **Outbox pattern** or **Saga pattern**.

swipe

Manish Jaiswal

EF Core Series

06

Outbox Pattern

```
public async Task SaveOrderAsync(Order order)
{
    using (var transaction = await _dbContext.Database.BeginTransactionAsync())
    {
        _dbContext.Orders.Add(order);
        await _dbContext.SaveChangesAsync();

        var outboxMessage = new OutboxMessage
        {
            Id = Guid.NewGuid(),
            OccurredOn = DateTime.UtcNow,
            Type = nameof(OrderCreated),
            Data = JsonConvert.SerializeObject(new OrderCreated(order.Id))
        };

        _dbContext.OutboxMessages.Add(outboxMessage);
        await _dbContext.SaveChangesAsync();

        await transaction.CommitAsync();
    }
}
```

swipe

Manish Jaiswal

EF Core Series

07

Saga Pattern

```
1   public async Task HandleOrderCreationAsync(Order order)
2   {
3       // Step 1: Create order
4       _dbContext.Orders.Add(order);
5       await _dbContext.SaveChangesAsync();
6
7       // Step 2: Publish event for inventory service
8       await _eventPublisher.PublishAsync(new OrderCreatedEvent(order.Id));
9   }
10
11  public async Task HandleOrderCreatedEventAsync(OrderCreatedEvent @event)
12  {
13      // Step 3: Update inventory based on the order created
14      var order = await _dbContext.Orders.FindAsync(@event.OrderId);
15      // Update inventory logic
16  }
```

swipe

Manish Jaiswal

EF Core Series

MASTERING

EF Core Applications with Azure

swipe

Manish Jaiswal

EF Core Series

01

Deploying EF Core Applications to Azure

- Simplify your deployment process using Azure App Service or Azure Kubernetes Service (AKS).
- Automate deployment pipelines with Azure DevOps.

swipe

Manish Jaiswal

EF Core Series

02

Example: **Basic Azure App Service Deployment Script**

```
# Sample deployment script using Azure CLI
az webapp create --resource-group MyResourceGroup \
                 --plan MyAppServicePlan \
                 --name MyWebApp \
                 --runtime "DOTNETCORE|8.0"

dotnet publish -c Release -o ./publish

az webapp deploy --resource-group MyResourceGroup \
                 --name MyWebApp \
                 --src-path ./publish
```

swipe

Manish Jaiswal

EF Core Series

03

Using Azure SQL with EF Core

- Leverage Azure SQL Database for robust and scalable relational data storage.
- Secure your data with built-in high availability and backup solutions.

swipe

Manish Jaiswal

EF Core Series

04

Example: Configuring EF Core with Azure SQL

```
1  public class ApplicationDbContext : DbContext
2  {
3      protected override void OnConfiguring(
4          DbContextOptionsBuilder optionsBuilder)
5      {
6          optionsBuilder.UseSqlServer(
7              "Server=tcp:myserver.database.windows.net,1433;" +
8              "Initial Catalog=mydb;" +
9              "User ID=myuser;" +
10             "Password=mypassword;");
11     }
12 }
```

swipe

Manish Jaiswal

EF Core Series

05

Best Practices for EF Core in Azure

- Optimize your database access with connection pooling and retry policies.
- Implement caching strategies using Azure Cache for Redis to reduce database load.

swipe

06

Example: Adding Retry Policy

```
public class ApplicationDbContext : DbContext
{
    protected override void OnConfiguring(
        DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(
            "Server=tcp:myserver.database.windows.net,1433;" +
            "Initial Catalog=mydb;" +
            "User ID=myuser;" +
            "Password=mypassword;",
            sqlOptions => sqlOptions.EnableRetryOnFailure(
                maxRetryCount: 5,
                maxRetryDelay: TimeSpan.FromSeconds(30),
                errorNumbersToAdd: null));
    }
}
```

swipe

Manish Jaiswal

EF Core Series

MASTERING

Real-World Examples with EF Core

swipe

Manish Jaiswal

EF Core Series

01

Case Study: Optimizing Performance for a High-Traffic E-Commerce Platform

In this case study, an e-commerce platform struggled with performance issues due to inefficient queries. By refactoring LINQ queries and utilizing async operations, the team achieved a 50% improvement in response times.

swipe

Manish Jaiswal

EF Core Series

02

Example

```
1  // Before optimization
2  var orders = context.Orders
3              .Where(o => o.CustomerId == customerId).ToList();
4
5  // After optimization
6  var orders = await context.Orders
7      .AsNoTracking()
8      .Where(o => o.CustomerId == customerId)
9      .ToListAsync();
10
```

swipe

Manish Jaiswal

EF Core Series

03

Success Story: Seamless Data Migration in a Financial Application

A financial application required a seamless data migration to a new schema. By leveraging EF Core's migrations and custom scripts, the team ensured a smooth transition without data loss or downtime.

swipe

Manish Jaiswal

EF Core Series

04

Example

```
// Applying a migration
dotnet ef migrations add UpdateSchema
dotnet ef database update

// Custom migration script
public override void Up()
{
    // Custom logic for data transformation
}
```

swipe

Manish Jaiswal

EF Core Series

05

Common Pitfall: Inefficient Loading of Related Data

A common mistake is loading related data inefficiently, leading to performance bottlenecks. Use eager loading, explicit loading, or lazy loading appropriately to optimize your queries.

swipe

06

Example

```
1  // Inefficient loading (N+1 problem)
2  var customers = context.Customers.ToList();
3  foreach (var customer in customers)
4  {
5      var orders = customer.Orders.ToList(); // N+1 queries
6  }
7
8  // Efficient loading with eager loading
9  var customersWithOrders = context.Customers
10     .Include(c => c.Orders)
11     .ToList();
```

swipe

Manish Jaiswal

EF Core Series

07

Lesson Learned: Importance of Query Optimization

Real projects highlight the importance of query optimization. Analyzing execution plans and indexing strategies can significantly improve performance.

swipe

Manish Jaiswal

EF Core Series

08

Example

```
1  // Adding an index to improve query performance
2  modelBuilder.Entity<Order>()
3      .HasIndex(o => o.OrderDate);
4
5  // Optimized query using index
6  var recentOrders = context.Orders
7      .Where(o => o.OrderDate > DateTime.Now.AddDays(-30))
8      .ToList();
9
```

swipe

Manish Jaiswal

EF Core Series

09

Case Study: Handling Concurrency Conflicts in a Collaborative App

A collaborative application faced concurrency issues with multiple users editing the same data. Implementing optimistic concurrency control helped manage conflicts effectively.

swipe

Manish Jaiswal

EF Core Series

10

Example

```
// Configuring concurrency token
modelBuilder.Entity<Document>()
    .Property(d => d.RowVersion)
    .IsRowVersion();

// Handling concurrency conflict
try
{
    context.SaveChanges();
}
catch (DbUpdateConcurrencyException ex)
{
    // Resolve conflict
}
```

swipe

Manish Jaiswal

EF Core Series

11

Common Pitfall: Ignoring Logging and Monitoring

Neglecting logging and monitoring can lead to undetected issues in production. Use EF Core's built-in logging and third-party tools to track query performance and errors.

swipe

Manish Jaiswal

EF Core Series

12

Example

```
// Enabling logging in EF Core
var optionsBuilder = new DbContextOptionsBuilder<MyDbContext>();
optionsBuilder.UseLoggerFactory(MyLoggerFactory);

// Configuring logging in appsettings.json
{
  "Logging": {
    "LogLevel": {
      "Microsoft.EntityFrameworkCore": "Information"
    }
  }
}
```

swipe

Manish Jaiswal

EF Core Series

13

Lesson Learned: Effective Use of Transactions

Managing transactions is crucial in real-world applications. Use EF Core's transaction management to ensure data integrity and handle rollback scenarios.

swipe

Manish Jaiswal

EF Core Series

14

Example

```
1   using var transaction = context.Database.BeginTransaction();
2   try
3   {
4       // Perform multiple operations
5       context.SaveChanges();
6       transaction.Commit();
7   }
8   catch
9   {
10      transaction.Rollback();
11      throw;
12  }
```

swipe