Database-First Approach in Entity Framework Core

1. Introduction to Database-First Approach

The Database-First approach in Entity Framework Core is a development workflow where you start with an existing database and generate the corresponding entity model (classes) based on that database schema. This approach is particularly useful in the following scenarios:

- Working with legacy databases that already exist
- When database design is handled by dedicated DBAs
- When database schema is designed using specialized database tools
- In organizations where the database is the source of truth
- When integrating with third-party databases that you don't control

In this lesson, we'll cover how to use Entity Framework Core's scaffolding tools to reverse-engineer an existing database into a set of entity classes and a DbContext.

2. Prerequisites

Before we start, make sure you have:

1. **Development Environment**:
   - Visual Studio 2019/2022 or Visual Studio Code
   - .NET 6.0+ SDK installed
2. **Required NuGet Packages**:

```
Microsoft.EntityFrameworkCore.Tools
Microsoft.EntityFrameworkCore.Design
```

3. **Database Provider Package** (choose one based on your database):

```
Microsoft.EntityFrameworkCore.SqlServer
Microsoft.EntityFrameworkCore.Sqlite
Microsoft.EntityFrameworkCore.PostgreSQL
Microsoft.EntityFrameworkCore.MySql
Oracle.EntityFrameworkCore
```

4. **Existing Database**:
   - A running database you can connect to
   - Connection string with appropriate permissions

3. Scaffolding Database with EF Core Tools

3.1 Using Package Manager Console (Visual Studio)

The Package Manager Console provides a convenient way to run Entity Framework commands within Visual Studio.

Basic Scaffolding Command

```
Scaffold-DbContext "Server=myServerAddress;Database=myDataBase;User Id=myUsername;Password=myPassword;"
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
```

This command:

- Connects to the specified database
- Uses the SQL Server provider
- Generates entity classes in the "Models" directory

Parameters Explained

- `"Connection String"` - The connection string to your database
- `Provider` - The database provider (e.g., Microsoft.EntityFrameworkCore.SqlServer)
- `-OutputDir` - Directory where the entity classes will be created
- `-Context` - Name of the generated DbContext class (optional)
- `-Schemas` - Database schemas to include (optional)
- `-Tables` - Specific tables to include (optional)

- `-Force` - Overwrite existing files (optional)
- `-NoOnConfiguring` - Don't generate DbContext.OnConfiguring method (useful for DI)
- `-UseDatabaseNames` - Use exact table/column names from database
- `-NoPluralize` - Don't pluralize or singularize entity and DbSet names
- `-ContextDir` - Directory where the DbContext class will be created

Examples with Different Options

Specifying DbContext Name:

```
Scaffold-DbContext "Connection String" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -Context
MyCustomDbContext
```

Scaffolding Specific Tables:

```
Scaffold-DbContext "Connection String" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -Tables Customer, Order,
OrderDetail
```

Scaffolding Specific Schema:

```
Scaffold-DbContext "Connection String" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -Schemas sales
```

Generating Only Data Models (No OnConfiguring):

```
Scaffold-DbContext "Connection String" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -NoOnConfiguring
```

Preserving Exact Database Names:

```
Scaffold-DbContext "Connection String" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -UseDatabaseNames
```

Putting DbContext in a Different Directory:

```
Scaffold-DbContext "Connection String" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -ContextDir Data -Context
ApplicationDbContext
```

Using Data Annotations Instead of Fluent API:

```
Scaffold-DbContext "Connection String" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -DataAnnotations
```

3.2 Using .NET CLI

If you're using Visual Studio Code or prefer the command line, you can use the .NET CLI to scaffold your database.

Basic Scaffolding Command

bash

```
dotnet ef dbcontext scaffold "Server=myServerAddress;Database=myDataBase;User Id=myUsername;Password=myPassword;"
Microsoft.EntityFrameworkCore.SqlServer --output-dir Models
```

Parameters Explained

The parameters are similar to the Package Manager Console, with slight syntax differences:

- `--output-dir` (instead of `-OutputDir`)
- `--context` (instead of `-Context`)
- `--schema` (instead of `-Schemas`)
- `--table` (instead of `-Tables`)
- `--force` (instead of `-Force`)
- `--no-onconfiguring` (instead of `-NoOnConfiguring`)
- `--use-database-names` (instead of `-UseDatabaseNames`)
- `--no-pluralize` (instead of `-NoPluralize`)
- `--context-dir` (instead of `-ContextDir`)
- `--data-annotations` (instead of `-DataAnnotations`)

Examples with Different Options

Specifying DbContext Name:

bash

```
dotnet ef dbcontext scaffold "Connection String" Microsoft.EntityFrameworkCore.SqlServer --output-dir Models --context
MyCustomDbContext
```

Scaffolding Specific Tables:

bash

```
dotnet ef dbcontext scaffold "Connection String" Microsoft.EntityFrameworkCore.SqlServer --output-dir Models --table
Customer --table Order --table OrderDetail
```

Scaffolding Specific Schema:

bash

```
dotnet ef dbcontext scaffold "Connection String" Microsoft.EntityFrameworkCore.SqlServer --output-dir Models --schema
sales
```

Generating Only Data Models (No OnConfiguring):

bash

```
dotnet ef dbcontext scaffold "Connection String" Microsoft.EntityFrameworkCore.SqlServer --output-dir Models --no-
onconfiguring
```

Preserving Exact Database Names:

bash

```
dotnet ef dbcontext scaffold "Connection String" Microsoft.EntityFrameworkCore.SqlServer --output-dir Models --use-
database-names
```

4. Connection String Security

It's important not to hard-code connection strings with sensitive information in your commands. Here are better approaches:

Using User Secrets (Development)

1. Initialize user secrets for your project:

bash

```
dotnet user-secrets init
```

2. Add your connection string to user secrets:

bash

```
dotnet user-secrets set "ConnectionStrings:DefaultConnection" "Server=myServerAddress;Database=myDataBase;User
Id=myUsername;Password=myPassword;"
```

3. Scaffold using the secret:

bash

```
dotnet ef dbcontext scaffold Name=ConnectionStrings:DefaultConnection Microsoft.EntityFrameworkCore.SqlServer --output-dir
Models
```

Using Environment Variables

1. Set an environment variable:

bash

```
# Windows
set ConnectionString="Server=myServerAddress;Database=myDataBase;User Id=myUsername;Password=myPassword;"

# macOS/Linux
export ConnectionString="Server=myServerAddress;Database=myDataBase;User Id=myUsername;Password=myPassword;"
```

2. Scaffold using the environment variable:

bash

```
dotnet ef dbcontext scaffold "%ConnectionString%" Microsoft.EntityFrameworkCore.SqlServer --output-dir Models
```

5. Step-by-Step Example: Scaffolding a SQL Server Database

Let's walk through a complete example of scaffolding the Northwind database:

5.1 Creating a New Project

bash

```
dotnet new webapi -n NorthwindApi
cd NorthwindApi
```

5.2 Adding Required Packages

bash

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

5.3 Scaffolding the Database

bash

```
dotnet ef dbcontext scaffold "Server=localhost;Database=Northwind;Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer --output-dir Models --context-dir Data --context NorthwindContext --no-
onconfiguring
```

5.4 Examining the Generated Code

After running the command, you'll find:

- Entity classes in the `Models` directory
- A `NorthwindContext` class in the `Data` directory

The `NorthwindContext` class will include:

- DbSet properties for each table
- Entity configurations in the `OnModelCreating` method
- Relationship definitions

Example of a Generated Entity Class:

csharp

```
// Models/Customer.cs
using System;
using System.Collections.Generic;

namespace NorthwindApi.Models
{
    public partial class Customer
    {
        public Customer()
        {
            Orders = new HashSet<Order>();
        }

        public string CustomerId { get; set; }
        public string CompanyName { get; set; }
```

```
        public string ContactName { get; set; }
        public string ContactTitle { get; set; }
        public string Address { get; set; }
        public string City { get; set; }
        public string Region { get; set; }
        public string PostalCode { get; set; }
        public string Country { get; set; }
        public string Phone { get; set; }
        public string Fax { get; set; }

        public virtual ICollection<Order> Orders { get; set; }
    }
}
```

Example of Generated DbContext:

csharp

```
// Data/NorthwindContext.cs
using System;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata;
using NorthwindApi.Models;

namespace NorthwindApi.Data
{
    public partial class NorthwindContext : DbContext
    {
        public NorthwindContext()
        {
        }

        public NorthwindContext(DbContextOptions<NorthwindContext> options)
            : base(options)
        {
        }

        public virtual DbSet<Category> Categories { get; set; }
        public virtual DbSet<Customer> Customers { get; set; }
        public virtual DbSet<Employee> Employees { get; set; }
        public virtual DbSet<Order> Orders { get; set; }
        public virtual DbSet<OrderDetail> OrderDetails { get; set; }
        public virtual DbSet<Product> Products { get; set; }
        public virtual DbSet<Supplier> Suppliers { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Customer>(entity =>
            {
                entity.HasKey(e => e.CustomerId);

                entity.Property(e => e.CustomerId)
                    .HasColumnName("CustomerID")
                    .HasMaxLength(5)
                    .IsFixedLength();

                entity.Property(e => e.Address).HasMaxLength(60);

                entity.Property(e => e.City).HasMaxLength(15);

                entity.Property(e => e.CompanyName)
                    .IsRequired()
                    .HasMaxLength(40);

                entity.Property(e => e.ContactName).HasMaxLength(30);

                entity.Property(e => e.ContactTitle).HasMaxLength(30);

                entity.Property(e => e.Country).HasMaxLength(15);

                entity.Property(e => e.Fax).HasMaxLength(24);

                entity.Property(e => e.Phone).HasMaxLength(24);

                entity.Property(e => e.PostalCode).HasMaxLength(10);

                entity.Property(e => e.Region).HasMaxLength(15);
            });
```

```
            // Additional entity configurations...

            OnModelCreatingPartial(modelBuilder);
        }

        partial void OnModelCreatingPartial(ModelBuilder modelBuilder);
    }
}
```

5.5 Registering the DbContext with Dependency Injection

Now you need to register the generated DbContext with the DI container in your `Program.cs`:

csharp

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container
builder.Services.AddControllers();

// Register DbContext
builder.Services.AddDbContext<NorthwindContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

// Configure Swagger/OpenAPI
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();

app.Run();
```

5.6 Adding Connection String to appsettings.json

json

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=localhost;Database=Northwind;Trusted_Connection=True;"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

6. Working with Different Database Providers

6.1 PostgreSQL

Install Package:

bash

```
dotnet add package Npgsql.EntityFrameworkCore.PostgreSQL
```

Scaffold Command:

bash

```
dotnet ef dbcontext scaffold "Host=localhost;Database=mydatabase;Username=myuser;Password=mypassword;"
Npgsql.EntityFrameworkCore.PostgreSQL --output-dir Models
```

6.2 MySQL

Install Package:

bash

```
dotnet add package Pomelo.EntityFrameworkCore.MySql
```

Scaffold Command:

bash

```
dotnet ef dbcontext scaffold "Server=localhost;Database=mydatabase;User=myuser;Password=mypassword;"
Pomelo.EntityFrameworkCore.MySql --output-dir Models
```

6.3 SQLite

Install Package:

bash

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

Scaffold Command:

bash

```
dotnet ef dbcontext scaffold "Data Source=mydatabase.db" Microsoft.EntityFrameworkCore.Sqlite --output-dir Models
```

6.4 Oracle

Install Package:

bash

```
dotnet add package Oracle.EntityFrameworkCore
```

Scaffold Command:

bash

```
dotnet ef dbcontext scaffold "User Id=myuser;Password=mypassword;Data Source=MyOracleDB;" Oracle.EntityFrameworkCore --
output-dir Models
```

7. Customizing Generated Code

7.1 Using Partial Classes

The generated entity classes are defined as `partial`, allowing you to extend them without modifying the generated code:

csharp

```
// Models/Customer.partial.cs
using System;

namespace NorthwindApi.Models
{
    public partial class Customer
    {
        // Add custom properties, methods, or validation logic
        public string FullAddress => $"{Address}, {City}, {PostalCode}, {Country}";

        public bool IsValidCustomer()
        {
            return !string.IsNullOrEmpty(CompanyName) && !string.IsNullOrEmpty(ContactName);
        }
```

```
    }
}
```

7.2 Using Partial Methods

The generated DbContext includes a partial method hook:

csharp

```
partial void OnModelCreatingPartial(ModelBuilder modelBuilder);
```

You can implement this method in a separate file:

csharp

```
// Data/NorthwindContext.partial.cs
using Microsoft.EntityFrameworkCore;
using NorthwindApi.Models;

namespace NorthwindApi.Data
{
    public partial class NorthwindContext
    {
        partial void OnModelCreatingPartial(ModelBuilder modelBuilder)
        {
            // Add custom configurations
            modelBuilder.Entity<Customer>()
                .HasIndex(c => c.CompanyName)
                .HasName("IX_Customers_CompanyName");

            // Add global query filters
            modelBuilder.Entity<Order>()
                .HasQueryFilter(o => !o.IsDeleted);
        }
    }
}
```

8. Re-Scaffolding and Updates

When the database schema changes, you'll need to update your entity model. There are two approaches:

8.1 Full Re-Scaffolding

Use the same command with the `-Force` (or `--force`) flag to overwrite existing files:

```
Scaffold-DbContext "Connection String" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -Force
```

or with .NET CLI:

bash

```
dotnet ef dbcontext scaffold "Connection String" Microsoft.EntityFrameworkCore.SqlServer --output-dir Models --force
```

Note: This will overwrite all your customizations, so use with caution!

8.2 Selective Updates

A better approach is to:

1. Create a temporary project
2. Scaffold the updated database there
3. Compare the differences
4. Manually update your entity classes

9. Best Practices for Database-First Approach

9.1 Keep Customizations Separate

- Use partial classes and methods for customizations
- Avoid modifying the generated files directly
- Consider creating a separate layer of DTOs above the entity model

9.2 Handling Updates

- Document your database schema
- Use version control for both database schema and generated code
- Consider using database migration tools for schema changes

9.3 Security Considerations

- Don't include sensitive connection strings in your code
- Use environment-specific configurations
- Consider using managed identity or connection string encryption in production

9.4 Architecture Recommendations

- Place generated entities in a dedicated Data or Infrastructure layer
- Use repositories or services to encapsulate data access logic
- Don't expose entity models directly in your API

10. Practical Exercise: Building a REST API with Database-First Approach

10.1 Scaffolding the Database

Use the commands from Section 5 to scaffold your database.

10.2 Creating a Repository Layer

csharp

```
// Repositories/ICustomerRepository.cs
public interface ICustomerRepository
{
    Task<IEnumerable<Customer>> GetAllAsync();
    Task<Customer> GetByIdAsync(string id);
    Task<Customer> CreateAsync(Customer customer);
    Task UpdateAsync(Customer customer);
    Task DeleteAsync(string id);
}

// Repositories/CustomerRepository.cs
public class CustomerRepository : ICustomerRepository
{
    private readonly NorthwindContext _context;

    public CustomerRepository(NorthwindContext context)
    {
        _context = context;
    }

    public async Task<IEnumerable<Customer>> GetAllAsync()
    {
        return await _context.Customers.ToListAsync();
    }

    public async Task<Customer> GetByIdAsync(string id)
    {
        return await _context.Customers
            .Include(c => c.Orders)
            .FirstOrDefaultAsync(c => c.CustomerId == id);
    }

    public async Task<Customer> CreateAsync(Customer customer)
    {
        _context.Customers.Add(customer);
        await _context.SaveChangesAsync();
        return customer;
    }

    public async Task UpdateAsync(Customer customer)
    {
        _context.Entry(customer).State = EntityState.Modified;
        await _context.SaveChangesAsync();
    }

    public async Task DeleteAsync(string id)
    {
```

```
        var customer = await _context.Customers.FindAsync(id);
        if (customer != null)
        {
            _context.Customers.Remove(customer);
            await _context.SaveChangesAsync();
        }
    }
}
```

10.3 Creating DTOs and Service Layer

csharp

```
// DTOs/CustomerDto.cs
public class CustomerDto
{
    public string Id { get; set; }
    public string CompanyName { get; set; }
    public string ContactName { get; set; }
    public string ContactTitle { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string Country { get; set; }
    public string Phone { get; set; }
}

// Services/ICustomerService.cs
public interface ICustomerService
{
    Task<IEnumerable<CustomerDto>> GetAllCustomersAsync();
    Task<CustomerDto> GetCustomerByIdAsync(string id);
    Task<CustomerDto> CreateCustomerAsync(CustomerDto customerDto);
    Task UpdateCustomerAsync(string id, CustomerDto customerDto);
    Task DeleteCustomerAsync(string id);
}

// Services/CustomerService.cs
public class CustomerService : ICustomerService
{
    private readonly ICustomerRepository _customerRepository;

    public CustomerService(ICustomerRepository customerRepository)
    {
        _customerRepository = customerRepository;
    }

    public async Task<IEnumerable<CustomerDto>> GetAllCustomersAsync()
    {
        var customers = await _customerRepository.GetAllAsync();
        return customers.Select(MapToDto);
    }

    public async Task<CustomerDto> GetCustomerByIdAsync(string id)
    {
        var customer = await _customerRepository.GetByIdAsync(id);
        if (customer == null) return null;
        return MapToDto(customer);
    }

    public async Task<CustomerDto> CreateCustomerAsync(CustomerDto customerDto)
    {
        var customer = MapToEntity(customerDto);

        // Generate new ID if not provided
        if (string.IsNullOrEmpty(customer.CustomerId))
        {
            customer.CustomerId = GenerateCustomerId();
        }

        await _customerRepository.CreateAsync(customer);
        return MapToDto(customer);
    }

    public async Task UpdateCustomerAsync(string id, CustomerDto customerDto)
    {
        var customer = await _customerRepository.GetByIdAsync(id);
        if (customer == null) throw new KeyNotFoundException($"Customer with ID {id} not found");
```

```
        // Update properties
        customer.CompanyName = customerDto.CompanyName;
        customer.ContactName = customerDto.ContactName;
        customer.ContactTitle = customerDto.ContactTitle;
        customer.Address = customerDto.Address;
        customer.City = customerDto.City;
        customer.Country = customerDto.Country;
        customer.Phone = customerDto.Phone;

        await _customerRepository.UpdateAsync(customer);
    }

    public async Task DeleteCustomerAsync(string id)
    {
        await _customerRepository.DeleteAsync(id);
    }

    private CustomerDto MapToDto(Customer customer)
    {
        return new CustomerDto
        {
            Id = customer.CustomerId,
            CompanyName = customer.CompanyName,
            ContactName = customer.ContactName,
            ContactTitle = customer.ContactTitle,
            Address = customer.Address,
            City = customer.City,
            Country = customer.Country,
            Phone = customer.Phone
        };
    }

    private Customer MapToEntity(CustomerDto dto)
    {
        return new Customer
        {
            CustomerId = dto.Id,
            CompanyName = dto.CompanyName,
            ContactName = dto.ContactName,
            ContactTitle = dto.ContactTitle,
            Address = dto.Address,
            City = dto.City,
            Country = dto.Country,
            Phone = dto.Phone
        };
    }

    private string GenerateCustomerId()
    {
        // Generate a 5-character ID (Northwind format)
        return new string(Enumerable.Range(0, 5)
            .Select(_ => (char)('A' + new Random().Next(0, 26)))
            .ToArray());
    }
}
```

10.4 Creating API Controllers

csharp

```
// Controllers/CustomersController.cs
[ApiController]
[Route("api/[controller]")]
public class CustomersController : ControllerBase
{
    private readonly ICustomerService _customerService;
    private readonly ILogger<CustomersController> _logger;

    public CustomersController(ICustomerService customerService, ILogger<CustomersController> logger)
    {
        _customerService = customerService;
        _logger = logger;
    }

    [HttpGet]
    public async Task<ActionResult<IEnumerable<CustomerDto>>> GetCustomers()
    {
        var customers = await _customerService.GetAllCustomersAsync();
```

```
        return Ok(customers);
    }

    [HttpGet("{id}")]
    public async Task<ActionResult<CustomerDto>> GetCustomer(string id)
    {
        var customer = await _customerService.GetCustomerByIdAsync(id);
        if (customer == null) return NotFound();
        return Ok(customer);
    }

    [HttpPost]
    public async Task<ActionResult<CustomerDto>> CreateCustomer(CustomerDto customerDto)
    {
        var createdCustomer = await _customerService.CreateCustomerAsync(customerDto);
        return CreatedAtAction(nameof(GetCustomer), new { id = createdCustomer.Id }, createdCustomer);
    }

    [HttpPut("{id}")]
    public async Task<IActionResult> UpdateCustomer(string id, CustomerDto customerDto)
    {
        try
        {
            await _customerService.UpdateCustomerAsync(id, customerDto);
            return NoContent();
        }
        catch (KeyNotFoundException)
        {
            return NotFound();
        }
    }

    [HttpDelete("{id}")]
    public async Task<IActionResult> DeleteCustomer(string id)
    {
        await _customerService.DeleteCustomerAsync(id);
        return NoContent();
    }
}
```

10.5 Registering Services

Finally, register your services in `Program.cs`:

csharp

```
// Register repositories and services
builder.Services.AddScoped<ICustomerRepository, CustomerRepository>();
builder.Services.AddScoped<ICustomerService, CustomerService>();
```

11. Comparing Database-First and Code-First Approaches

When working with Entity Framework Core, understanding the differences between Database-First and Code-First approaches is crucial for choosing the right strategy for your project. Let's examine both approaches across multiple dimensions to help you make an informed decision.

11.1 Conceptual Differences

Database-First:

- Starts with an existing database schema
- Generates C# classes from database tables
- Database is the source of truth
- Changes to the model come from database changes

Code-First:

- Starts with C# domain classes
- Generates database schema from code
- Code is the source of truth
- Changes to the database come from code changes

11.2 Side-by-Side Comparison

Feature	Database-First	Code-First
Starting Point	Existing database	C# domain classes
Control Over	Entity model	Database schema and entity model
Schema Changes	Made in database, then re-scaffold	Made in code, then generate migrations
Workflow	Database → Scaffold → Code	Code → Migration → Database
Learning Curve	Easier if you're a database expert	Easier if you're a C# developer
Version Control	Database changes harder to track	All changes in code, easy to track
Team Collaboration	DBA and developer roles clearly separated	Developers need more database knowledge
Complex Mappings	May require custom code	Can be expressed in Fluent API or attributes
Database Support	Limited to what providers support	Full support for all EF Core features

11.3 When to Use Database-First

Best Scenarios:

1. Working with legacy or existing databases
2. When database schema is controlled by DBAs
3. When integrating with third-party databases
4. In organizations with strict database governance
5. When database design is done with specialized tools

Advantages:

1. Leverage existing database investments
2. No need to define schema in code
3. Quick to get started with existing data
4. Clear separation between database and application teams
5. Database optimizations are preserved

Challenges:

1. Changes require re-scaffolding
2. Custom modifications may be lost during re-scaffolding
3. Less control over entity model design
4. May generate unnecessary navigation properties
5. Limited flexibility for complex models

11.4 When to Use Code-First

Best Scenarios:

1. New projects with no existing database
2. Domain-driven design approaches
3. When developers control both code and database
4. When using Test-Driven Development
5. Microservices and container-based applications

Advantages:

1. Full control over both entity model and database schema
2. Database changes tracked in version control via migrations
3. Domain model can be designed without database concerns
4. Easy to create test databases
5. Better support for agile development practices

Challenges:

1. Requires more knowledge of EF Core conventions
2. May produce less optimized database schemas
3. More effort to map to an existing database
4. Can be complex to handle certain database features
5. Might require more maintenance for migrations

11.5 Hybrid Approaches

In practice, you can combine elements of both approaches:

1. **Selective Scaffolding:**
   - Start with Database-First for existing tables
   - Use Code-First for new features
2. **Initial Scaffolding with Code Evolution:**
   - Scaffold existing database once
   - Move to Code-First for ongoing development
3. **Schema Compare Tools:**
   - Use database schema compare tools to sync changes
   - Apply changes to both model and database directly

11.6 Real-World Example: Migrating Legacy Application

Let's look at a scenario where a team is modernizing a legacy application:

Step 1: Use Database-First to scaffold the existing database

bash

```
dotnet ef dbcontext scaffold "Connection String" Microsoft.EntityFrameworkCore.SqlServer --output-dir Models
```

Step 2: Create a snapshot of the current model as a migration

bash

```
dotnet ef migrations add InitialCreate
```

Step 3: Switch to Code-First for new features

csharp

```
public partial class LegacyContext
{
    // Generated by Database-First scaffolding
}

// New entities added with Code-First
public class Feature
{
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime CreatedDate { get; set; }
    // ...
}
```

Step 4: Add a migration for the new entity

bash

```
dotnet ef migrations add AddFeatureEntity
```

This hybrid approach allows teams to work with existing data while moving toward a more maintainable Code-First approach over time.

11.7 Decision Matrix

Use this decision matrix to help choose the right approach:

Factor	Choose Database-First If...	Choose Code-First If...
Database Existence	Database already exists	Starting with a new database
Team Structure	Separate DBA and developer teams	Developers handle both code and database
Design Focus	Database design is primary	Domain model design is primary
Maintenance Control	Database changes controlled externally	Full control over all changes
Project Type	Integration or reporting projects	New applications with complex domain logic
Timeline	Need quick start with existing database	Can invest time in proper domain modeling
Version Control	Database has its own versioning system	Need all changes in code repository

12. Conclusion

The Database-First approach in Entity Framework Core provides a powerful way to work with existing databases. By following the steps and best practices outlined in this lesson, you can effectively:

1. Scaffold database schema into C# classes
2. Customize and extend the generated code
3. Build maintainable applications on top of existing databases
4. Keep your entity model synchronized with database changes

While Code-First approach gives you more control over your entity design, Database-First is often the practical choice when working with established databases or in organizations where database design is handled separately from application development.

Database-First allows you to leverage the power of Entity Framework Core's ORM capabilities while respecting the existing database schema as the source of truth.

The choice between Database-First and Code-First is not absolute—many projects benefit from a pragmatic combination of both approaches based on specific requirements and constraints. By understanding the strengths and limitations of each approach, you can make informed decisions that best serve your development process and project goals.