Complete SQL and EF Core Guide for C# Developers

A comprehensive guide to mastering SQL fundamentals and applying database concepts in C# applications using Entity Framework Core, with Azure SQL considerations.

Introduction

This document provides a complete roadmap for C# developers to master database concepts from foundational SQL through advanced Entity Framework Core implementation. The focus is on Microsoft's T-SQL dialect because Azure SQL Database, Managed Instance, and Synapse SQL all use the SQL Server T-SQL dialect with minor cloud-specific differences.

Part 1: Relational Database Fundamentals

Relational Modeling

Why: A solid schema avoids anomalies, speeds queries, and simplifies code. Understanding entities, relationships, keys, and normalization is essential before writing queries.

When/Where: Before app coding and migrations; revisit when new features add tables/relations or performance issues suggest redesign.

Example: Customers(1-many) Orders relationship

- Primary key on Customers.CustomerID
- Foreign key Orders.CustomerID references Customers.CustomerID
- Unique index on Orders(OrderNumber)

Data Definition Language (DDL)

Why: DDL defines structure, constraints, and defaults the optimizer can use. Stricter DDL equals safer data and simpler queries.

When/Where: Early in projects; evolve via migrations; avoid instance-level options on Azure SQL (use contained users and service objectives).

Example:

```
CREATE TABLE Customers (
  CustomerID INT IDENTITY PRIMARY KEY,
  Name NVARCHAR(100) NOT NULL,
  Email NVARCHAR(255) NOT NULL UNIQUE,
  City NVARCHAR(80) NULL
);

CREATE TABLE Orders (
  OrderID INT IDENTITY PRIMARY KEY,
  CustomerID INT NOT NULL,
```

```
  OrderDate DATE NOT NULL DEFAULT (GETDATE()),
  OrderNumber NVARCHAR(30) NOT NULL UNIQUE,
  CONSTRAINT FK_Orders_Customers
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

Part 2: SQL Querying Essentials

Basic Querying (SELECT)

Why: Everyday reads for APIs and reports; good habits scale to analytics and ELT.

When/Where: Everywhere—APIs, jobs, reports; start with selective predicates and projection to minimize IO.

Example:

```
-- Find last 30 days' orders with customer name
SELECT o.OrderID, o.OrderDate, c.Name
FROM Orders o
JOIN Customers c ON c.CustomerID = o.CustomerID
WHERE o.OrderDate &gt;= DATEADD(day, -30, CAST(GETDATE() AS date))
ORDER BY o.OrderDate DESC;
```

Filtering, Sorting, and Paging

Why: Shape result sets for UI and services; prevent over-fetching.

When/Where: List endpoints; dashboards; background exports.

Example:

```
SELECT o.OrderID, o.OrderDate, c.Name
FROM Orders o
JOIN Customers c ON c.CustomerID = o.CustomerID
WHERE c.City = @City
ORDER BY o.OrderDate DESC
OFFSET @Skip ROWS FETCH NEXT @Take ROWS ONLY;
```

Joins and Set Operations

Why: Relational power; replace N+1 lookups; combine streams efficiently.

When/Where: Reporting, aggregates, cross-entity reads.

Example:

```
-- Customers with/without orders
SELECT c.CustomerID, c.Name, COUNT(o.OrderID) AS OrdersCount
FROM Customers c
LEFT JOIN Orders o ON o.CustomerID = c.CustomerID
GROUP BY c.CustomerID, c.Name
HAVING COUNT(o.OrderID) &gt;= 0;
```

Aggregation and Window Functions

Why: Summarize data and compute running metrics without extra roundtrips.

When/Where: KPI cards, financial rollups, pagination with totals.

Example:

```
-- Month revenue and running total
SELECT
  FORMAT(o.OrderDate,'yyyy-MM') AS Month,
  SUM(oi.Quantity * oi.UnitPrice) AS Revenue,
  SUM(SUM(oi.Quantity * oi.UnitPrice)) OVER
    (ORDER BY FORMAT(o.OrderDate,'yyyy-MM')) AS RunningRevenue
FROM Orders o
JOIN OrderItems oi ON oi.OrderID = o.OrderID
GROUP BY FORMAT(o.OrderDate,'yyyy-MM');
```

Part 3: Data Integrity and Performance

Constraints and Data Integrity

Why: Prevent bad data at the source; simplifies app logic.

When/Where: Always—PK, FK, UNIQUE, CHECK, NOT NULL; use CHECK for domain rules.

Example:

```
ALTER TABLE OrderItems
ADD CONSTRAINT CK_OrderItems_PositiveQty CHECK (Quantity &gt; 0),
    CONSTRAINT CK_OrderItems_PositivePrice CHECK (UnitPrice &gt;= 0);
```

Indexing Fundamentals

Why: Orders of magnitude faster reads; the main lever for latency and cost.

When/Where: After observing query patterns; add nonclustered indexes on predicates/joins; include columns to cover queries.

Example:

```
-- Frequent filter on CustomerID and OrderDate with projection
CREATE INDEX IX_Orders_CustomerID_OrderDate
  ON Orders(CustomerID, OrderDate DESC)
  INCLUDE (OrderNumber);
```

Key Indexing Guidelines:

- Clustered index on primary key (automatic)
- Nonclustered indexes on frequently filtered columns
- Composite indexes with most selective column first

- Include columns for covering indexes
- Monitor index usage and remove unused indexes

Transactions and Concurrency

Why: Keep data consistent across multi-step operations; avoid partial writes.

When/Where: Order placements, money movement, inventory updates; choose isolation based on conflict risk.

Example:

```
BEGIN TRAN;
  INSERT INTO Orders(CustomerID, OrderDate, OrderNumber)
  VALUES (@CustomerID, GETDATE(), @OrderNumber);

  INSERT INTO OrderItems(OrderID, ProductID, Quantity, UnitPrice)
  SELECT SCOPE_IDENTITY(), ProductID, Quantity, Price
  FROM @CartItems;

COMMIT;
```

Isolation Levels:

- READ UNCOMMITTED: Fastest, allows dirty reads
- READ COMMITTED: Default, prevents dirty reads
- REPEATABLE READ: Prevents dirty and non-repeatable reads
- SERIALIZABLE: Strictest, prevents all anomalies

Part 4: Advanced SQL Concepts

Stored Procedures and Scripts

Why: Encapsulate data logic close to data; reduce chatty calls; allow plan stability.

When/Where: Complex writes (MERGE/UPSERT), batch ops, data quality checks; invoked by ADF/Logic Apps/EF.

Example:

```
CREATE OR ALTER PROCEDURE dbo.UpsertCustomer
  @Email NVARCHAR(255),
  @Name NVARCHAR(100),
  @City NVARCHAR(80) = NULL
AS
BEGIN
  SET NOCOUNT ON;
  MERGE Customers AS target
  USING (SELECT @Email AS Email, @Name AS Name, @City AS City) AS src
  ON target.Email = src.Email
  WHEN MATCHED THEN
    UPDATE SET Name = src.Name, City = src.City
  WHEN NOT MATCHED THEN
```

```
    INSERT (Email, Name, City) VALUES (src.Email, src.Name, src.City);
END;
```

Performance Patterns

Why: Right access paths and minimal scans reduce CPU/IO and cloud costs.

When/Where: Add covering indexes; use sargable predicates; inspect plans; avoid SELECT * in hot paths.

Key Performance Tips:

- Use parametrized queries to enable plan reuse
- Avoid functions in WHERE clauses
- Use EXISTS instead of IN with subqueries
- Limit result sets with TOP or paging
- Use appropriate data types (avoid implicit conversions)

Part 5: Azure SQL Considerations

T-SQL Differences in Azure SQL

Most T-SQL works the same in SQL Server and Azure SQL, but some differences exist:

Not Supported in Azure SQL Database:

- Database snapshots
- FILESTREAM and FILETABLE
- SQL Server Agent (use Elastic Jobs instead)
- Cross-database queries (use Elastic Query)
- Some system stored procedures

Azure SQL Specific Features:

- Service tiers and compute sizes
- Automatic tuning
- Elastic pools
- Built-in backup and point-in-time restore

Part 6: Entity Framework Core Implementation

EF Core Overview

Why: Productivity with LINQ, change tracking, and migrations; balances speed and maintainability for most app CRUD.

When/Where: Standard line-of-business apps and services; augment with Dapper/ADO.NET for hot paths or bulk ops if needed.

Project Setup

Steps: Define entity classes (POCOs), DbContext, connection string; add migrations; update database; use DI for context lifetime.

Entity Classes:

```
public class Customer
{
    public int CustomerID { get; set; }
    public string Email { get; set; } = "";
    public string Name { get; set; } = "";
    public string? City { get; set; }
    public List&lt;Order&gt; Orders { get; set; } = new();
}

public class Order
{
    public int OrderID { get; set; }
    public int CustomerID { get; set; }
    public DateTime OrderDate { get; set; }
    public string OrderNumber { get; set; } = "";
    public Customer Customer { get; set; } = null!;
    public List&lt;OrderItem&gt; Items { get; set; } = new();
}

public class OrderItem
{
    public int OrderItemID { get; set; }
    public int OrderID { get; set; }
    public int ProductID { get; set; }
    public int Quantity { get; set; }
    public decimal UnitPrice { get; set; }
    public Order Order { get; set; } = null!;
}
```

DbContext Configuration:

```
public class AppDbContext : DbContext
{
    public DbSet&lt;Customer&gt; Customers =&gt; Set&lt;Customer&gt;();
    public DbSet&lt;Order&gt; Orders =&gt; Set&lt;Order&gt;();
    public DbSet&lt;OrderItem&gt; OrderItems =&gt; Set&lt;OrderItem&gt;();

    public AppDbContext(DbContextOptions&lt;AppDbContext&gt; options) : base(options) {}

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity&lt;Customer&gt;()
            .HasIndex(x =&gt; x.Email)
            .IsUnique();

        modelBuilder.Entity&lt;Order&gt;()
            .HasOne(o =&gt; o.Customer)
            .WithMany(c =&gt; c.Orders)
```

```
            .HasForeignKey(o =&gt; o.CustomerID)
            .OnDelete(DeleteBehavior.Restrict);

        modelBuilder.Entity&lt;OrderItem&gt;()
            .HasOne(oi =&gt; oi.Order)
            .WithMany(o =&gt; o.Items)
            .HasForeignKey(oi =&gt; oi.OrderID)
            .OnDelete(DeleteBehavior.Cascade);

        modelBuilder.Entity&lt;OrderItem&gt;()
            .Property(oi =&gt; oi.UnitPrice)
            .HasColumnType("decimal(18,2)");
    }
}
```

Dependency Injection Setup

Program.cs Configuration:

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

builder.Services.AddDbContext&lt;AppDbContext&gt;(options =&gt;
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

var app = builder.Build();
```

Connection String (appsettings.json):

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=localhost;Database=ShopDb;Trusted_Connection=True;TrustSe
  }
}
```

Migrations and Schema Evolution

Why: Versioned, repeatable schema changes; CI/CD friendly; scriptable for DBAs.

Common Commands:

```
# Add initial migration
dotnet ef migrations add InitialCreate

# Update database
dotnet ef database update

# Generate SQL script
dotnet ef migrations script

# Add new migration
dotnet ef migrations add AddProductTable
```

```
# Remove last migration (if not applied)
dotnet ef migrations remove
```

Migration Best Practices:

- Always review generated migration code
- Add custom SQL for data transformations
- Use nullable columns for non-breaking changes
- Plan rollback strategies for production deployments

Querying with EF Core

Why: Type-safe queries; provider-translated to SQL; easy projection.

Basic Queries:

```
// Simple filter and projection
var recentOrders = await context.Orders
    .Where(o =&gt; o.OrderDate &gt;= DateTime.UtcNow.AddDays(-30))
    .Select(o =&gt; new { o.OrderID, o.OrderDate, o.Customer.Name })
    .ToListAsync();

// Include related data
var ordersWithItems = await context.Orders
    .Include(o =&gt; o.Items)
    .Include(o =&gt; o.Customer)
    .Where(o =&gt; o.CustomerID == customerId)
    .OrderByDescending(o =&gt; o.OrderDate)
    .ToListAsync();

// Pagination
var pagedOrders = await context.Orders
    .OrderByDescending(o =&gt; o.OrderDate)
    .Skip(skip)
    .Take(take)
    .AsNoTracking()
    .ToListAsync();
```

Advanced Queries:

```
// Group by with aggregation
var monthlyRevenue = await context.OrderItems
    .GroupBy(oi =&gt; oi.Order.OrderDate.ToString("yyyy-MM"))
    .Select(g =&gt; new
    {
        Month = g.Key,
        Revenue = g.Sum(oi =&gt; oi.Quantity * oi.UnitPrice),
        OrderCount = g.Select(oi =&gt; oi.OrderID).Distinct().Count()
    })
    .ToListAsync();

// Complex joins
var customerStats = await context.Customers
```

```
    .Select(c =&gt; new
    {
        c.Name,
        c.Email,
        TotalOrders = c.Orders.Count(),
        TotalSpent = c.Orders.SelectMany(o =&gt; o.Items)
            .Sum(oi =&gt; oi.Quantity * oi.UnitPrice),
        LastOrderDate = c.Orders.Max(o =&gt; o.OrderDate)
    })
    .Where(cs =&gt; cs.TotalOrders &gt; 0)
    .ToListAsync();
```

Data Modification Operations

Insert Operations:

```
// Simple insert
var customer = new Customer
{
    Name = "John Doe",
    Email = "john.doe@example.com",
    City = "New York"
};

context.Customers.Add(customer);
await context.SaveChangesAsync();

// Insert with related data
var order = new Order
{
    CustomerID = customerId,
    OrderDate = DateTime.UtcNow,
    OrderNumber = Guid.NewGuid().ToString("N")[..10],
    Items = cartItems.Select(ci =&gt; new OrderItem
    {
        ProductID = ci.ProductId,
        Quantity = ci.Quantity,
        UnitPrice = ci.Price
    }).ToList()
};

context.Orders.Add(order);
await context.SaveChangesAsync();
```

Update Operations:

```
// Tracked entity update
var customer = await context.Customers.FindAsync(customerId);
if (customer != null)
{
    customer.Name = updatedName;
    customer.City = updatedCity;
    await context.SaveChangesAsync();
}
```

```
// Untracked update
context.Customers.Update(new Customer
{
    CustomerID = customerId,
    Name = updatedName,
    Email = email,
    City = updatedCity
});
await context.SaveChangesAsync();

// Bulk update (EF Core 7+)
await context.Customers
    .Where(c =&gt; c.City == "Old City")
    .ExecuteUpdateAsync(s =&gt; s.SetProperty(c =&gt; c.City, "New City"));
```

Delete Operations:

```
// Single delete
var order = await context.Orders.FindAsync(orderId);
if (order != null)
{
    context.Orders.Remove(order);
    await context.SaveChangesAsync();
}

// Bulk delete (EF Core 7+)
await context.Orders
    .Where(o =&gt; o.OrderDate &lt; DateTime.UtcNow.AddYears(-2))
    .ExecuteDeleteAsync();
```

Transaction Management

Why: Coordinate multi-entity writes; set isolation where needed.

Example:

```
await using var transaction = await context.Database.BeginTransactionAsync();
try
{
    // Multiple operations
    var customer = new Customer { Name = "Test", Email = "test@test.com" };
    context.Customers.Add(customer);
    await context.SaveChangesAsync();

    var order = new Order
    {
        CustomerID = customer.CustomerID,
        OrderDate = DateTime.UtcNow,
        OrderNumber = "ORD001"
    };
    context.Orders.Add(order);
    await context.SaveChangesAsync();
```

```
    await transaction.CommitAsync();
}
catch
{
    await transaction.RollbackAsync();
    throw;
}
```

Stored Procedure Integration

Why: Reuse DB-side logic for upserts/batches; leverage plans and permissions.

Calling Stored Procedures:

```
// Execute stored procedure with parameters
var emailParam = new SqlParameter("@Email", emailValue);
var nameParam = new SqlParameter("@Name", nameValue);
var cityParam = new SqlParameter("@City", (object?)cityValue ?? DBNull.Value);

await context.Database.ExecuteSqlRawAsync(
    "EXEC dbo.UpsertCustomer @Email, @Name, @City",
    emailParam, nameParam, cityParam);

// Execute with return values
var results = await context.Customers
    .FromSqlRaw("EXEC dbo.GetCustomersByCity @City",
        new SqlParameter("@City", cityName))
    .ToListAsync();
```

Performance Optimization

Why: Keep LINQ efficient; avoid implicit client evaluation; reduce tracking overhead.

Optimization Patterns:

```
// Use AsNoTracking for read-only queries
var readOnlyOrders = await context.Orders
    .AsNoTracking()
    .Where(o =&gt; o.OrderDate &gt;= startDate)
    .ToListAsync();

// Project to DTOs to reduce data transfer
var orderSummaries = await context.Orders
    .Select(o =&gt; new OrderSummaryDto
    {
        OrderId = o.OrderID,
        OrderNumber = o.OrderNumber,
        CustomerName = o.Customer.Name,
        TotalAmount = o.Items.Sum(i =&gt; i.Quantity * i.UnitPrice)
    })
    .ToListAsync();

// Use compiled queries for frequently executed queries
private static readonly Func&lt;AppDbContext, int, Task&lt;Customer?&gt;&gt; GetCustomerBy
```

```
    EF.CompileAsyncQuery((AppDbContext context, int id) =&gt;
        context.Customers.FirstOrDefault(c =&gt; c.CustomerID == id));

// Usage
var customer = await GetCustomerById(context, customerId);

// Split queries for large includes
var ordersWithAllData = await context.Orders
    .AsSplitQuery()
    .Include(o =&gt; o.Customer)
    .Include(o =&gt; o.Items)
    .ToListAsync();
```

Code-First vs Database-First

Code-First Approach:

- Faster iteration, migrations in code
- Best for greenfield and domain-driven designs
- Full control over entity relationships and constraints

Database-First Approach:

- Align with existing databases
- Scaffold models from schema
- Good for legacy or DBA-owned databases

Scaffolding Example:

```
dotnet ef dbcontext scaffold "ConnectionString" Microsoft.EntityFrameworkCore.SqlServer -o
```

Part 7: Azure Deployment Considerations

Azure SQL Deployment

Key Differences:

- Use contained users instead of SQL logins
- Configure service tier and compute size
- Set up firewall rules for client access
- Use Azure Key Vault for connection strings

Connection String for Azure SQL:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=tcp:myserver.database.windows.net,1433;Initial Catalog=my
  }
}
```

Migration Deployment Strategies

Development Environment:

```
// Automatic migration in development
if (app.Environment.IsDevelopment())
{
    using var scope = app.Services.CreateScope();
    var context = scope.ServiceProvider.GetRequiredService&lt;AppDbContext&gt;();
    await context.Database.MigrateAsync();
}
```

Production Environment:

- Generate SQL scripts: `dotnet ef migrations script`
- Apply via Azure DevOps pipelines or manual deployment
- Use idempotent scripts for safe repeated execution

Part 8: Practical Implementation Guide

8-Week Learning Plan

Weeks 1-2: SQL Fundamentals

- Practice DDL/DML for shop schema
- Add constraints and relationships
- Master joins, aggregations, and window functions
- Experiment with indexes and query plans

Week 3: EF Core Setup

- Create .NET API project
- Configure DbContext and entities
- Implement code-first migrations
- Build basic CRUD operations

Week 4: Advanced EF Core

- Add complex queries with LINQ
- Implement stored procedure calls
- Optimize with AsNoTracking and projections
- Profile generated SQL queries

Weeks 5-6: Performance and Indexing

- Analyze execution plans
- Add strategic indexes
- Implement caching strategies

- Compare EF Core vs. Dapper for performance

Weeks 7-8: Azure Integration

- Deploy to Azure SQL Database
- Configure authentication and firewall
- Set up CI/CD for migrations
- Implement monitoring and alerts

Sample Project Architecture

```
ShopAPI/
├── Controllers/
│   ├── CustomersController.cs
│   └── OrdersController.cs
├── Data/
│   ├── AppDbContext.cs
│   ├── Entities/
│   └── Migrations/
├── Services/
│   ├── CustomerService.cs
│   └── OrderService.cs
├── Models/
│   └── DTOs/
└── Program.cs
```

Best Practices Summary

SQL Best Practices:

- Always use parameterized queries
- Design indexes based on query patterns
- Implement proper constraint validation
- Use transactions for multi-step operations
- Monitor and optimize query performance

EF Core Best Practices:

- Use AsNoTracking for read-only queries
- Project to DTOs to minimize data transfer
- Implement proper error handling and logging
- Use migrations for schema changes
- Configure proper entity relationships

Azure Best Practices:

- Use Managed Identity for authentication
- Implement retry logic for transient failures

- Monitor with Application Insights
- Use service tiers appropriate for workload
- Implement backup and disaster recovery

Conclusion

Mastering SQL design, querying, constraints, indexing, transactions, and stored procedures in tandem with EF Core yields robust, maintainable, and performant data applications. The T-SQL compatibility between SQL Server and Azure SQL Database, along with well-documented cloud differences, provides a smooth path for deploying applications to Azure with minimal friction.

This comprehensive approach ensures that developers can leverage both the productivity benefits of EF Core and the performance advantages of well-designed SQL, creating applications that scale effectively in cloud environments while maintaining code quality and development velocity.