



Week 5 - C# NOTES

LINQ

Language-Integrated Query, is a powerful feature in C# that allows us to write queries for data collections like arrays, lists, or databases in a more readable and expressive way.

Linq is executed two ways

1. Immediate: Immediate execution means that the data source is read and the operation is performed once. Ex: Sum, Average, Count, First, Max
2. Deferred: Deferred execution means that the operation isn't performed at the point in the code where the query is declared. The operation is performed only when the query variable is enumerated, for example by using a `foreach` statement.

To make deferred result immediate use `ToList()` or `ToArray()`

There is two way to write linq query

Method way

Ex:

```
var booksAfter1950Method = books.Where(b => b.PublicationYear > 1950)
    .Join(authors,
        b => b.AuthorId,
        a => a.Id,
        (b, a) => new
        {
            BookTitle = b.Title,
            AuthorName = a.Name,
            Year = b.PublicationYear
        })
```

```
    })  
    .OrderBy(result ⇒ result.Year);
```

Query way

Ex:

```
var booksAfter1950Query = from b in books  
    join a in authors on b.AuthorId equals a.Id  
    where b.PublicationYear > 1950  
    orderby b.PublicationYear  
    select new  
    {  
        BookTitle = b.Title,  
        AuthorName = a.Name,  
    };
```

Operations

Filter

```
List<int> nums = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
List<int> odds = nums.Where(n ⇒ n % 2 == 1).ToList();
```

Select

```
List<string> strings = new List<string> { "apple", "banana", "cherry", "date", "fig", "grape" };  
List<int> lens = strings.Select(s ⇒ s.Length).ToList();
```

Join

```

var booksAfter1950Method = books.Where(b => b.PublicationYear > 1950)
    .Join(authors,
        b => b.AuthorId,
        a => a.Id,
        (b, a) => new
        {
            BookTitle = b.Title,
            AuthorName = a.Name,
            Year = b.PublicationYear
        })
    .OrderBy(result => result.Year);

```

Group by

```

var departmentStatsMethod = employees.GroupBy(d => d.Field<int>("DepartmentID"))
    .Join(departments,
        empGroup => empGroup.Key,
        d => d.Field<int>("DepartmentID"),
        (empGroup, d) => new {
            DepartmentName = d.Field<string>("Department
Name"),
            EmployeeCount = empGroup.Count(),
            AverageSalary = empGroup.Average(e => e.Field<
decimal>("Salary")),
            MaximumSalary = empGroup.Sum(e => e.Field<d
ecimal>("Salary"))
        });

```

Order by, pagination

```

var pagedEmployeeMethod = employees.OrderByDescending(e => e.Field<
decimal>("Salary"))
    .Skip(1)

```

```
.Take(2)
.Select(e => new
{
    Name = e.Field<string>("Name"),
    Salary = e.Field<decimal>("Salary")
});
```

EF Core

Library: Microsoft.EntityFrameworkCore

To connect with MySQL use nuget package:

Pomelo.EntityFrameworkCore.MySql

To use CLI to make migration use package:

Microsoft.EntityFrameworkCore.Tools

DB first approach

First need to Scaffold-dbcontext

```
Scaffold-DbContext "Server=localhost;user=root;password=*****;data
base=employee" Pomelo.EntityFrameworkCore.MySql -OutputDir Models
```

It will create Models folder and model class in it

It will also create DbContext class. An instance of `DbContext` represents a session with the database which can be used to query and save instances of entities to a database.

Code first approach

Create models

Create a context class inherit it from DbContext

Register dbsets

```
public DbSet<Student> Students { get; set; }
```

override OnConfiguring method

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder.UseMySQL("Server=localhost;user=root;password=*****;
    database=student", ServerVersion.AutoDetect("Server=localhost;user=root;password=*****;database=employee"));
}
```

Override OnModelCreating method to define relations

```
protected override void OnModelCreating(ModelBuilder modelBuilder){}
```

To establish one to many relationship

In one side model

```
public int DepartmentId { get; set; }
[ForeignKey("DepartmentId")]
public Department Department { get; set; }
```

In many side model

```
public ICollection<Student> Students { get; set; }
```

In model creating method

```

modelBuilder.Entity<Student>()
    .HasOne(s ⇒ s.Department)
    .WithMany(d ⇒ d.Students)
    .HasForeignKey(s ⇒ s.DepartmentId)
    .OnDelete(DeleteBehavior.Restrict);

```

To establish many to many relation

models

student model

```

public ICollection<Course> Courses { get; set; }

```

course model

```

public ICollection<Student> Students { get; set; }

```

OnModelCreating method

```

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

```

Migrations

Migrations: A **migration** is a way to **incrementally update the database schema** to keep it in sync with your **C# model classes**.

To create a migration

```

Add-Migration <migration_name>

```

It will create a new file in Migration folder which have information about new updates It have two methods Up(To apply migration) and Down(To revert migration)

To apply migration to db

```
Update-Database [<MigrationName>]
```

We can use migration name to apply specific migration to db else last migration will be applied

To remove migration

```
Remove-Migration
```

It will remove the last migration

CRUD

Read

We can query data using LINQ

```
var students = context.Students
    .Where(s => s.DepartmentId == computerDepartment.Id)
    .Include(s => s.Courses)
    .Select(s => new
    {
        StudentName = s.Name,
        Course = s.Courses.Select(c => c.Title).ToList(),
        Department = s.Department.Name
    });
```

Update

```
var data4 = context.T01s.FirstOrDefault(x => x.T01f01 == 7);
if (data4 != null)
{
    data4.T01f02 = "iOS";
    context.SaveChanges();
}
```

Delete

```
var data3 = context.T01s.FirstOrDefault(x => x.T01f01 == 7);
context.T01s.Remove(data3);
context.SaveChanges();
```

Insert

```
T01 t01 = new T01()
{
    T01f01 = 7,
    T01f02 = "Android",
    T01f03 = "anand"
};
context.T01s.Add(t01);
context.SaveChanges();
```

SP

If it returns result first we need to create a DTO for sp's result and add it in context class

```
string departmentName = "Engineering";
List<GetEmployeeByDepartmentResult> data5 = context.GetEmployeeByD
```



```

departmentResults.FromSqlInterpolated($"CALL get_employee_by_department({departmentName})").ToList();

foreach (GetEmployeeByDepartmentResult item in data5)
{
    Console.WriteLine($"{item.T02F01} {item.T02F02} {item.T02F03}");
}

```

If it does not return a table we can use

```

context.Database.ExecuteSqlRaw();

```

Transaction

```

using (var transaction = context.Database.BeginTransaction())
{
    context.Database.ExecuteSqlRaw("Delete from t03");

    transaction.Rollback();
}

```

To commit

```

transactin.Commit();

```

To create savepoint

```

transaction.CreateSavePoint(name);

```

To rollback to savepoint

```

transaction.RollbackToSavepoint(name);

```

Security

Hashing

To create SHA256 hash

```
using (SHA256 sha256 = SHA256.Create())
{
    byte[] bytes = sha256.ComputeHash(System.Text.Encoding.UTF8.GetBytes(input));
    return Convert.ToHexString(bytes);
}
```

However we should use a random salt while hashing password and store it alongside with hashedpassword by concating it

To hash password

Saltsize should be 16byte (128bit)

Hashsize should be 32byte (256bit)

Here iterations should be more that 10000 to make hashing slower so attacker can't **brute force** passwords. This is defined in pbkdf2 standard which is implemented by Rfc2898DeriveBytes

```
public static string HashPassword(string password)
{
    // Generate a salt
    byte[] salt = RandomNumberGenerator.GetBytes(SaltSize);

    // Derive the hash
    var pbkdf2 = new Rfc2898DeriveBytes(password, salt, Iterations, HashAlgorithmName.SHA256);
}
```

```

byte[] hash = pbkdf2.GetBytes(HashSize);

// Combine salt and hash
byte[] hashedBytes = new byte[SaltSize + HashSize];
Array.Copy(salt, 0, hashedBytes, 0, SaltSize);
Array.Copy(hash, 0, hashedBytes, SaltSize, HashSize);

// Return as base64 string
return Convert.ToBase64String(hashedBytes);
}

```

To check password

Here we are using FixedTimeEquals because it compare each byte even if find that bytes are not matching so it prevents attacker to make time analysis attack.

Time analysis attack: Attacker try to find return time and try to analyze which is first byte which does not match and get some insights

```

public static bool VerifyPassword(string password, string storedHash)
{
    // Extract the bytes
    byte[] hashedBytes = Convert.FromBase64String(storedHash);

    // Get the salt
    byte[] salt = new byte[SaltSize];
    Array.Copy(hashedBytes, 0, salt, 0, SaltSize);

    // Get the hash
    var pbkdf2 = new Rfc2898DeriveBytes(password, salt, Iterations, HashAlgorithmName.SHA256);
    byte[] hash = pbkdf2.GetBytes(HashSize);

    // Compare the results
    // Use a time-constant comparison to prevent timing attacks
    return CryptographicOperations.FixedTimeEquals(hash, hashedBytes.As

```

```
Span(SaltSize));  
}
```

AES (Symmetric hashing)

Use AesGcm as it ensure integrity and confidentiality

Use random key and IV(Initialization vector)

Tag is used to ensure integrity it throw error if cipher text is altered

Encryption

```
byte[] key = RandomNumberGenerator.GetBytes(32); // 256 bit key  
byte[] nonce = RandomNumberGenerator.GetBytes(AesGcm.Nonces.MaxSize);  
byte[] cipherText = new byte[plainTextBytes.Length];  
byte[] tag = new byte[AesGcm.Tags.MaxSize]; // Authentication tag  
  
aesGcm.Encrypt(nonce, plainTextBytes, cipherText, tag);  
  
byte[] forStorage = new byte[nonce.Length + tag.Length + cipherText.Length];  
Array.Copy(nonce, 0, forStorage, 0, nonce.Length);  
Array.Copy(tag, 0, forStorage, nonce.Length, tag.Length);  
Array.Copy(cipherText, 0, forStorage, nonce.Length + tag.Length, cipherText.Length);
```

Decryption

```

byte[] extractedNonce = new byte[AesGcm.NonceByteSizes.MaxSize];
byte[] extractedTag = new byte[AesGcm.TagByteSizes.MaxSize];
byte[] extractedCipherText = new byte[forStorage.Length - extractedNonce.Length - extractedTag.Length];

Array.Copy(forStorage, 0, extractedNonce, 0, extractedNonce.Length);
Array.Copy(forStorage, extractedNonce.Length, extractedTag, 0, extractedTag.Length);
Array.Copy(forStorage, extractedNonce.Length + extractedTag.Length, extractedCipherText, 0, extractedCipherText.Length);

byte[] decryptedBytes = new byte[extractedCipherText.Length];

aesGcm.Decrypt(extractedNonce, extractedCipherText, extractedTag, decryptedBytes);

string decrypted = Encoding.UTF8.GetString(decryptedBytes);

```

RSA

Rsa is used for asymmetric key cryptography

Rsa is slow it should only use to encrypt small data, mostly keys

For large data use AES

Encrypt

```

static byte[] RsaEncrypt(string plainText, RSAParameters publicKey)
{
    using(RSACryptoServiceProvider rsa = new RSACryptoServiceProvider())
    {
        rsa.ImportParameters(publicKey);
        byte[] bytes = Encoding.UTF8.GetBytes(plainText);
    }
}

```

```

        return rsa.Encrypt(bytes, false);
    }
}

```

Decrypt

```

static byte[] RsaDecrypt(byte[] cipherText, RSAParameters privateKey)
{
    using(RSACryptoServiceProvider rsa = new RSACryptoServiceProvider(
    ))
    {
        rsa.ImportParameters(privateKey);
        return rsa.Decrypt(cipherText, false);
    }
}

```

To generate random keys

```

using(RSACryptoServiceProvider rsa = new RSACryptoServiceProvider(2048))
{
    RSAParameters publicKey = rsa.ExportParameters(false);
    RSAParameters privateKey = rsa.ExportParameters(true);
}

```

Digital Signature

DS is used to ensure integrity

```

using(RSA rsa = RSA.Create(2048))
{
    byte[] data = Encoding.UTF8.GetBytes(original);
}

```

```

byte[] sign = rsa.SignData(data, HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1);

Console.WriteLine($"Sign: {Convert.ToBase64String(sign)}");

bool isVerified = rsa.VerifyData(data, sign, HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1);
Console.WriteLine($"Signature verification status : {isVerified}");
}

```

Best practise

1. Never use own crypto use standard, well-known, battle-tested algorithms like System.Security.Cryptography
2. Use modern algorithm like AES-256, RSA, SHA-256m PBKDF2
3. Securely manage keys:

Never hard code keys

Don't store it in configuration files

Ex: use Secret Manager

which store json file in separate file in users/appdata so it never track by version control

To access it

```

var configuration = builder.Build();
Console.WriteLine($"Secret Key: {configuration["APIKEY"]}");

```

To use it while scaffold-dbcontext it will automatically find from configuration file and secrets file

```

scaffold-dbcontext "Name=DefaultConnection" Microsoft.EntityFrameworkCore.SqlServer -o Models

```

4. Use authenticated encryption: Like AesGcm which provide integrity with confidentiality
5. Salt password: Use unique, cryptographically random salt for each user. Store the salt alongside the hash in database. This prevents attackers from using pre-computed rainbow tables
6. For key use cryptographically secure random number like `System.Security.Cryptography.RandomNumberGenerator`

Dynamic Type

The *dynamic language runtime* (DLR) is a runtime environment that adds a set of services for dynamic languages to the common language runtime (CLR).

Dynamic type let's us use variable without declaring its type

Its type is decided at run time

It is different from object as object needs boxing and unboxing while dynamic type don't need it

But it doesn't provide compile time warnings or errors if type mismatch it will throw `RuntimeBinderException` at run time

Ex: It will work without exception

```
dynamic a = 5;  
a = "HELLO";
```

Usage

- **Interoperability with Dynamic Languages:** It's excellent for working with libraries from languages like Python or Ruby.
- **Working with COM Objects:** Interacting with technologies like Microsoft Office automation becomes much simpler.

- **Parsing Dynamic Data:** Handling data from sources like JSON or XML where the structure isn't known beforehand. You can navigate the parsed object without creating a rigid class structure first.
- **Simplifying Reflection:** It can sometimes provide a cleaner syntax than using complex reflection code to invoke methods dynamically.

Ex:

```
string jsonString = @"{
  'user': { 'id': 123, 'name': 'Alex', 'roles': ['Admin', 'Editor'] },
  'lastLogin': '2025-10-15T14:30:00Z'
}";

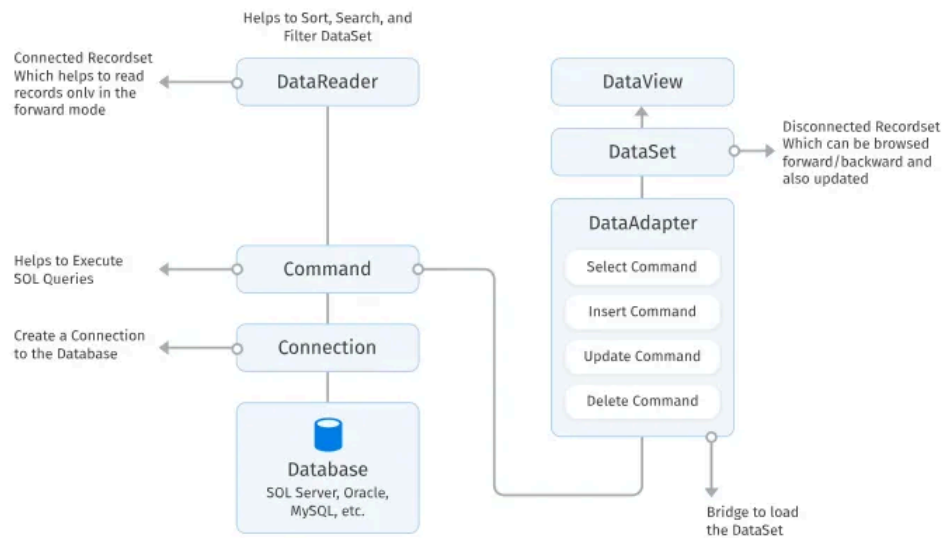
dynamic jsonObj = JsonConvert.DeserializeObject<dynamic>(jsonString);
Console.WriteLine($"User ID: {jsonObj.user.id}");
```

Limitation

- No compile time error checking
- performance overhead : DLR has to perform work at run time to figure out what we are trying to call
- Loss of intellisense and tooling support
- reduce code readability and maintainability

ADO.NET

Architecture of ADO.NET



DataReader is forward only, read only, stream that can be used to read row by row, it is connection oriented

DataAdapter uses cached data, it can be filled and updated to original DB

Use Command when

- **want to run a single query or command** (like `INSERT`, `UPDATE`, `DELETE`, or `SELECT`).
- **don't need to store or cache** results in memory.
- want **fine-grained control** over execution (parameters, transactions, etc.).

Use DataReader when

- need **fast and efficient reading** of large result sets.
- **only need to display** or process data — not modify it.
- can **keep the connection open** while reading.

Use DataAdapter when

- want **disconnected data access** (load data, close connection, work offline).
- need to **fetch, modify, and update** data back to the database.
- want **automatic generation of commands** (`InsertCommand` , `UpdateCommand` , `DeleteCommand`).

DataAdapter

To fill datatable

```
MySqlConnection connection = new MySqlConnection(connectionString);

MySqlDataAdapter adapter = new MySqlDataAdapter(query, connection);

MySqlCommandBuilder builder = new MySqlCommandBuilder(adapter);

connection.Open();

DataTable T01 = new DataTable();

adapter.Fill(T01);
```

Delete

```
foreach(DataRow row in T01.Rows)
{
    if ((string)row["T01F02"] == "iOS")
    {
        row.Delete();
    }
}
adapter.Update(T01);
```

Insert

```
T01.Rows.Add(10, "Android", "Anand");  
adapter.Update(T01);
```

Update

```
foreach (DataRow row in T01.Rows)  
{  
    if ((int)row["T01F01"] == 1)  
    {  
        row["T01F03"] = "Delhi";  
    }  
}  
adapter.Update(T01);
```

Read

```
foreach (DataRow row in T01.Rows)  
{  
    Console.WriteLine($"{row["T01F01"]} {row["T01F02"]} {row["T01F03"]}");  
}  
}
```

DataReader

```
MySqlConnection connection = new MySqlConnection(connectionString);  
  
MySqlCommand command = new MySqlCommand(query, connection);  
  
connection.Open();
```

```

MySQLDataReader reader = command.ExecuteReader();

if(reader.HasRows)
{
    while(reader.Read())
    {
        Console.WriteLine($"{reader.GetInt32(0)}, {reader.GetString(1)} {reader.GetString(2)}, {reader.GetString(3)}");
    }
}

```

Commands

Raw query

```

string insertQuery = "INSERT INTO T01(T01F01, T01F02, T01F03) VALUES
(@id, @name, @location)";

connection.Open();

MySQLCommand command = new MySQLCommand(insertQuery, connection);

command.Parameters.AddWithValue("@id", 11);
command.Parameters.AddWithValue("@name", "iOS");
command.Parameters.AddWithValue("@location", "delhi");

int rows = command.ExecuteNonQuery();

connection.Close();

Console.WriteLine($"Affected rows : {rows}");

```

SP

```
connection.Open();

MySQLCommand command2 = new MySQLCommand("GET_EMPLOYEE_BY_
DEPARTMENT", connection);

command2.CommandType = System.Data.CommandType.StoredProcedure;

command2.Parameters.AddWithValue("p_dept_name", "Engineering");

MySQLDataReader reader = command2.ExecuteReader();

while (reader.Read())
{
    Console.WriteLine($"{reader.GetInt32(0)} {reader.GetString(1)} {reader.G
etString(2)}");
}

connection.Close();
```

SP with out parameter

```
connection.Open();

MySQLCommand command3 = new MySQLCommand("count_emp", connect
ion);

command3.CommandType = System.Data.CommandType.StoredProcedure;

command3.Parameters.Add("p_count", MySQLDbType.Int32);
command3.Parameters["p_count"].Direction = System.Data.ParameterDire
ction.Output;
```

```
command3.ExecuteNonQuery();

connection.Close();

int count = (int)command3.Parameters["p_count"].Value;
Console.WriteLine(count);
```

Transaction

```
connection.Open();

MySQLTransaction transaction = connection.BeginTransaction();

try
{
    MySqlCommand insertCommand = new MySqlCommand("", connection,
transaction);
    insertCommand.CommandText = "DELETE FROM T01";

    throw new Exception("Somethind went wrong");

    transaction.Commit();
}
catch (Exception ex)
{
    transaction.Rollback();
    Console.WriteLine("Transaction Rollbacked" + ex.Message);
}

connection.Close();
```