**LINQ Practice Tasks**

1. **Basic Operations**
   * Retrieve all elements from a collection.

return await \_dbContext.Set<T>().ToListAsync();

* + Retrieve specific elements from a collection using where clause.

var students = await \_studentRepository.Table.Where(s => s.Age == age).ToListAsync();

* + Retrieve distinct elements from a collection.

var students = await \_studentRepository.Table.Where(s => s.Age == age).Distinct().ToListAsync();

1. **Projection**
   * Select specific fields from a collection.

var students = await \_studentRepository.Table

.Where(s => s.Age == age)

.Select(s => new StudentDTO

{

Name= s.StudentFirstName +s.StudentLastName,

Age = s.Age,

Class = s.Class,

Mail = s.Mail

})

.Distinct()

.ToListAsync();

return students ?? new List<StudentDTO>();

* + Select new objects from a collection with selected fields.

var studentDTOs = await \_studentRepository.Table

.Select(s => new StudentDTO

{

Name= s.StudentFirstName +s.StudentLastName,

StudentFatherName= s.StudentFatherName,

Age = s.Age,

Class = s.Class,

Mail = s.Mail

})

.ToListAsync();

return studentDTOs

## Perform calculations and select the results.

**Steps to Perform Calculations and Select Results**

1. **Summation**: Calculate the sum of a particular numeric field.

var ages = await \_studentRepository.Table.SumAsync(s => s.Age);

1. **Average**: Calculate the average of a numeric field.

var ages = await \_studentRepository.Table.AverageAsync(s => s.Age);

1. **Minimum and Maximum**: Find the minimum or maximum value of a numeric field.

var ages = await \_studentRepository.Table.MinAsync(s => s.Age);

var ages = await \_studentRepository.Table.MaxAsync(s => s.Age);

1. **Arithmetic Operations**: Perform arithmetic operations (addition, subtraction, multiplication, division) on fields.

var futureAges = students.Select(s => new { s.Name, FutureAge = s.Age + 5 });

1. **Create New Objects**: Create new objects with calculated fields.
2. **Ordering**
   * Sort elements in ascending order.

var students = await \_studentRepository.Table.OrderBy(s => s.Age ).ToListAsync();

* + Sort elements in descending order.

var students = await \_studentRepository.Table.OrderByDescending(s => s.Age ).ToListAsync();

* + Sort elements based on multiple fields.

var students = await \_studentRepository.Table.OrderByDescending(s => s.Age ).ThenByDescending(s => s.Class).ToListAsync();

# **Joining**

## Perform inner join between two collections.

public async Task<List<StudentDTO>> InnerJoin()

{

try

{

var departments= await \_departmentServices.GetAllAsync();

// Fetch the list of students from the database

var students = await \_studentRepository.Table.AsNoTracking().ToListAsync();

// Perform the join in-memory

var innerJoin = from student in students

join department in departments

on student.DepartmentId equals department.Id

select new

{

student.StudentFirstName,

student.Age,

department.DepartmenrDescription

};

// Map the results to StudentDTO

var result = innerJoin.Select(x => new StudentDTO

{

Name = x.StudentFirstName,

Age = x.Age,

Department = x.DepartmenrDescription

}).ToList();

return result;

}

catch (Exception ex)

{

\_logger.LogError(ex, "Error occurred while getting students by age.");

throw;

}

}

## Perform left outer join between two collections.

// Perform the left outer join in-memory

var leftOuterJoin = from student in students

join department in departments

on student.DepartmentId equals department.Id into deptGroup

from dept in deptGroup.DefaultIfEmpty() // This ensures that all students are included

select new

{

student.StudentFirstName,

student.Age,

DepartmentDescription = dept?.DepartmenrDescription

};

## Perform Right outer join between two collections.

// Perform the right outer join in-memory (by swapping the tables)

var rightOuterJoin = from department in departments

join student in students

on department.Id equals student.DepartmentId into studentGroup

from stud in studentGroup.DefaultIfEmpty() // This ensures that all departments are included

select new

{

StudentFirstName = stud?.StudentFirstName,

Age = stud?.Age,

DepartmentDescription = department.DepartmenrDescription

};

## Perform left Inner join between two collections.

var departmentDTO = await \_departmentServices.GetAllAsync();

var departments = \_mapper.Map<List<Department>>(departmentDTO);

var students = await \_studentRepository.Table.AsNoTracking().ToListAsync();

var innerJoin = students

.Join(departments,

student => student.DepartmentId,

department => department.Id,

(student, department) => new

{

student.StudentFirstName,

student.Age,

department.DepartmenrDescription

})

.ToList();

var result = innerJoin.Select(x => new StudentDTO

{

Name = x.StudentFirstName,

Age = x.Age,

Department = x.DepartmenrDescription

}).ToList();

return result;

## Perform Right Inner join between two collections.

var departmentDTO = await \_departmentServices.GetAllAsync();

var departments = \_mapper.Map<List<Department>>(departmentDTO);

var students = await \_studentRepository.Table.AsNoTracking().ToListAsync();

var innerJoin = departments

.Join(students,

department => department.Id,

student => student.DepartmentId,

(department, student) => new

{

student.StudentFirstName,

student.Age,

department.DepartmenrDescription

})

.ToList();

var result = innerJoin.Select(x => new StudentDTO

{

Name = x.StudentFirstName,

Age = x.Age,

Department = x.DepartmenrDescription

}).ToList();

return result;

* + Perform group join between two collections.

1. **Grouping**
   * Group elements by a specific field.

public async Task<List<GroupedStudentsDTO>> GroupByDepartment()

{

try

{

var students = await \_studentRepository.Table.AsNoTracking().ToListAsync();

var groupedStudents = students

.GroupBy(student => student.DepartmentId)

.Select(group => new GroupedStudentsDTO

{

DepartmentId = group.Key,

Students = group.Select(student => new StudentDTO

{

Name = student.StudentFirstName,

Age = student.Age

}).ToList()

})

.ToList();

return groupedStudents;

}

catch (Exception ex)

{

\_logger.LogError(ex, "Error occurred while grouping students by department.");

throw;

}

}

* + Group elements by multiple fields.

public async Task<List<GroupedStudentsDTO>> GroupByDepartmentAndAge()

{

try

{

var students = await \_studentRepository.Table.AsNoTracking().ToListAsync();

var groupedStudents = students

.GroupBy(student => new { student.DepartmentId, student.Age })

.Select(group => new GroupedStudentsDTO

{

DepartmentId = group.Key.DepartmentId,

Age = group.Key.Age,

Students = group.Select(student => new StudentDTO

{

Name = student.StudentFirstName,

Age = student.Age

}).ToList()

})

.ToList();

return groupedStudents;

}

catch (Exception ex)

{

\_logger.LogError(ex, "Error occurred while grouping students by department and age.");

throw;

}

}

* + Retrieve the count of elements in each group.

public async Task<List<GroupedStudentsCountDTO>> CountStudentsInEachDepartment()

{

try

{

var students = await \_studentRepository.Table.AsNoTracking().ToListAsync();

var studentCount = students

.GroupBy(student => student.DepartmentId)

.Select(group => new GroupedStudentsCountDTO

{

DepartmentId = group.Key,

StudentCount = group.Count()

})

.ToList();

return studentCount;

}

catch (Exception ex)

{

\_logger.LogError(ex, "Error occurred while counting students in each department.");

throw;

}

}

1. **Aggregation**
   * Calculate the sum of a field in a collection.
   * Calculate the average of a field in a collection.
   * Calculate the maximum and minimum of a field in a collection.
2. **Quantifiers**
   * Check if any element in a collection satisfies a condition.
   * Check if all elements in a collection satisfy a condition.
   * Check if a collection contains a specific element.
3. **Set Operations**
   * Retrieve the union of two collections.
   * Retrieve the intersection of two collections.
   * Retrieve the difference between two collections.
4. **Partitioning**
   * Retrieve the first N elements from a collection.
   * Skip the first N elements and retrieve the remaining elements from a collection.
   * Retrieve elements while a condition is true and then skip the rest.
5. **Element Operations**
   * Retrieve the first element from a collection.
   * Retrieve the first element that satisfies a condition.
   * Retrieve the single element that satisfies a condition.
6. **Conversion**
   * Convert a collection to a list.
   * Convert a collection to a dictionary.
   * Convert a collection to an array.
7. **Generation**
   * Create a collection of a specific size initialized with default values.
   * Create a collection with a range of values.
   * Create a collection with repeated values.
8. **Miscellaneous**
   * Use select many to flatten nested collections.
   * Retrieve the index of elements while querying a collection.
   * Combine multiple LINQ methods in a single query