Đồ Án Công Nghệ Mới Hệ Thống Thông Tin

Sinh viên thực hiện:

12C12010 – Nguyễn Thị Mai

19C12004 – Nguyễn Phan Chí Thành

Refactor ADO.NET in C#

Mục tiêu:

Xây dựng thư viện kết nối trên nhiều hệ cơ sở dữ liệu như Mysql, Postgresql, Oracle, Sql Server… . Thư viện có thể thực hiện được các yêu cầu sau:

- ✓ Thực hiện lấy kết quả từ truy vấn select trả về nhiều dòng dữ liệu.
- ✓ Thực hiện lấy kế quả từ truy vấn 1 dòng dữ liệu.
- ✓ Thực hiện lấy kết quả trả về khi thực hiện truy vấn từ nhiều bảng dữ liệu.
- ✓ Thực viện có thể lấy kết quả từ tham số trả về trong thủ tục nội tại.
- ✓ Kết quả thực hiện truy vấn có thể tùy biến trả về List, Object hoặc Dictionary
- ✓ Thực hiện thêm, xóa, sửa trên 1 dòng
- ✓ Thực hiện thêm, xóa , sửa nhiều dòng dữ liệu.

1. Thử nghiệm

Kết quả thực hiện trên cơ sở dữ liệu có lược đồ đơn giản như sau:

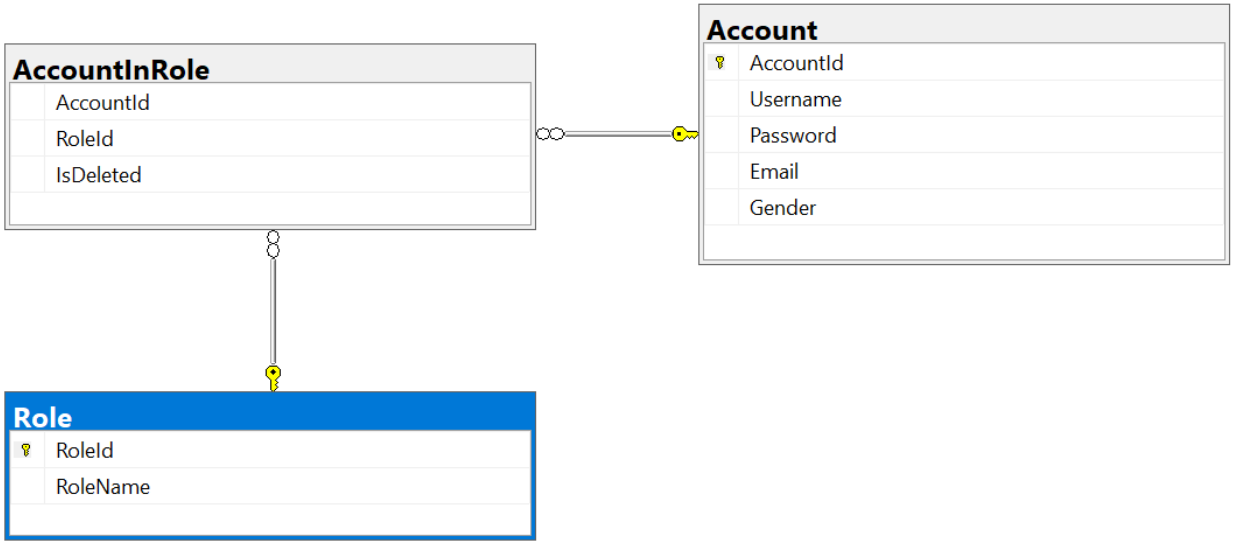


Figure 1: Lược đồ CSDL

2. Cách tiếp cận thông thường.

Để thực hiện lấy ra danh sách Role trong cơ sở dữ liệu ta thực hiện các bước sau

Khởi đầu ta tạo một lớp Role tương ứng với bảng Role trong cơ sở dữ liệu

```
public class Role
{
    1 reference
    public Guid Id { get; set; }
    1 reference
    public string Name { get; set; }
}
```

Figure 2: class Role

Tiếp theo ta tạo lớp **RoleRepository** trong lớp này ta tạo phương thức **GetRoles** để lấy ra danh sách **Role** trong bảng **Role**

```
public class RoleRepository
{
    readonly static string connectionString = "Data Source=.;Initial Catalog=BikeStore;Integrated Security=True";
    0 references
    public List<Role> GetRoles()
    {
        using (IDbConnection connection = new SqlConnection(connectionString))
        {
            using (IDbCommand command = connection.CreateCommand())
            {
                command.CommandText = "SELECT * FROM Role";
                connection.Open();
                using (IDataReader reader = command.ExecuteReader())
                {
                    List<Role> list = new List<Role>();
                    while (reader.Read())
                    {
                        list.Add(new Role
                        {
                            Id = (Guid)reader["RoleId"],
                            Name = (string)reader["RoleName"]
                        });
                    }
                    return list;
                }
            }
        }
    }
}
```

Figure 3: Method GetRoles

Trong hàm GetRoles ta nhận thấy nếu muốn thực hiện truy vấn trên bảng dữ liệu khác ta phải thực hiện lại tương tự như hàm này, đây là cách viết viết bị trùng lắp nhiều. Nếu tiếp cận theo cách này ta thấy rằng việc thay đổi code sẽ dẫn đến phải thay đổi hàng loạt các bảng tương ưng.

3. Tiếp cận tổng quát

3.1 Phân rã tổng quát

Nhìn một cách tổng quan ta có thể chia hàm **GetRoles** *ở Figure 3* phía trên thành 3 phần được cho bởi ba hình sau

Figure 4: Dependency Injection

Để có thể thực hiện được trên nhiều hệ quản trị CSDL khác nhau ta có thể sử dụng cơ chế Dependency injection. Cách thực hiện này khá dễ dàng bằng cách ta sử dụng 1 interface **IDbConnection** làm thuộc tính cho class **RoleRepository** sau đó class được truyền vào bằng constructor của class **RoleRepository**

Figure 5: ICommandCreator

Trong bước này để có thể tùy biến thực hiện các câu truy vấn khác nhau hoặc có thể gọi được thủ tục ta sẽ thay thế 3 dòng code trên bằng 1 interface **ICommandCreator**

Figure 6: ICommandCallback

Trên đoạn code được đánh dấu bằng hình vuông ta có thể thay thế bằng 1 interface **ICommandCallback** để có thể thực thi cả 3 trường hợp **ExecuteReader**, **ExecuteNonQuery**, **ExecuteScalar**.

3.2 Các bước thực hiện tạo interface

Từ bước phân tính tổng quát trên, ta tiến hàng xây dựng các interface thực hiện như sau

- ✓ Tạo interface **ICommandCreate**

```
public interface ICommandCreator
{
    1 reference
    IDbCommand CreateCommand(IDbConnection connection);
}
```

Figure 7: Interface ICommandCreate

Trong interface **ICommandCreator** có hàm **CreateCommand** với tham số đầu vào là 1 interface **IDbConnection** tương ứng tùy thuộc vào từng loại database ta có thể tạo command tương ứng để thực hiện trên từng loại database

- ✓ Tạo interface **ICommandCallback**

```
1 reference
public interface ICommandCallback<T>
{
    1 reference
    T Handle(IDbCommand command);
}
```

Figure 8: ICommandCallback

Trong Interface **ICommandCallback<T>** chỉ 1 hàm Handle với tham số đầu vào là 1 IDbCommand và được trả về là 1 kiểu dữ liệu bất kỳ tùy vào mục đích thực thi command có thể là ExecuteNonQuery thì sẽ được return về kiểu số nguyên, nếu là ExecuteScalar thì sẽ đươc trả về kiểu object hoặc thực hiện ExecuteReader thì 1 thể trả vể 1 lớp cụ thể hoặc trả về một List hoặc cũng có thể là 1 Dictionary.

3.3 Tạo interface dùng chung cho các class Repository

- ✓ Interface IDbContext

Tạo interface IDbContext dùng chung cho mọi truy vấn

```
1 reference
public interface IDbContext
{
}
```

Figure 9: Interface IDbContext

Trong interface IDbContext tạm thời lúc này ta chưa xác định được nên viết hàm gì cho hợp lý tạm thời chưa có phương thức nào.

- ✓ Class DbContext

Để cụ thể hóa cho interface **IDbContext** tao tạo lớp **DBContext** để cài đặt cho interface **IDbContext**

```
public class DbContext : IDbContext
{
    IDbConnection connection;
    0 references
    public DbContext(IDbConnection connection)
    {
        this.connection = connection;
    }
    0 references
    T Execute<T>(ICommandCreator creator, ICommandCallback<T> callback)
    {
        IDbCommand command = null;
        try
        {
            command = creator.CreateCommand(connection);
            T result = callback.Handle(command);
            return result;
        }
        finally
        {
            if (command != null)
            {
                command.Dispose();
            }
        }
    }
}
```

Figure 10: Class DbContext

Trong lớp **DbContext** để có thể thực hiện Dypendency injection ta sử dụng Constructor với tham số đầu vào là một interface **IDbconnection** để người sử dụng có thể tùy biến thay đổi kết nối đến nhiều CSDL khác nhau.

Trong lớp **DbContext** ra tạo một hàm Execute hàm này là một hàm quan trọng nó có thể thực hiện mọi câu truy vấn với 2 tham số đầu vào là 2 interface là **ICommandCreator** để khởi tạo command và **ICommandCallback** để thực hiện truy vấn dữ liệu.

- ✓ Class **SimpleCommandCreator**

Đặt vấn để vậy interface **ICommandCreator** sẽ thực hiện như thế nào, điều này được thực hiện bằng 1 lớp **SimpleCommandCreator** để cài đặt cụ thể cho interface **ICommandCreator**

```
public class SimpleCommandCreator : ICommandCreator
{
    string sql;
    CommandType commandType;
    0 references
    public SimpleCommandCreator(string sql, CommandType commandType = CommandType.Text)
    {
        this.sql = sql;
        this.commandType = commandType;
    }
    2 references
    public IDbCommand CreateCommand(IDbConnection connection)
    {
        IDbCommand command = connection.CreateCommand();
        command.CommandText = sql;
        command.CommandType = commandType;
        return command;
    }
}
```

Figure 11: Class SimpleCommandCreator

Trong lớp này người dùng có thể truyền vào bất kỳ một câu truy vấn vào thông qua tham số tham số và để có thể thực hiện truy vấn trực tiếp hay gọi thủ tục nội ta người dùng cần khai báo thêm 1 tham số là CommandType để chỉ thị cho thư viện ADO.NET của C# biết thực hiện trên thủ tục. Trong lớp **SimpleCommdCreator** ta cài đặt hàm cụ thể **CreateCommand** cụ thể cho interface **ICommandCreator**

- ✓ Phương thức **Execute** trong interface **IDbContext**

Quay trở lại interface **IDbContext** lúc này ta có thể thêm 1 hàm Execute

```
1 reference
public interface IDbContext
{
    1 reference
    T Execute<T>(string sql, ICommandCallback<T> callback, CommandType commandType = CommandType.Text);
}
```

Figure 12: Method Execute

Trong hàm **Execute** này người dùng muốn thực hiện cần truyền 2 tham số là 1 câu sql, và một lớp cần được cài đặt kế thừa từ interface **ICommandCallback**.

- ✓ Overide phương thức **Execute**

Lúc này trong lớp **DbContext** ta cài đặt hàm **Execute** rất dễ dàng như sau

```
1 reference
public class DbContext : IDbContext
{
    IDbConnection connection;
    0 references
    public DbContext(IDbConnection connection)
    {
        this.connection = connection;
    }

    1 reference
    public T Execute<T>(string sql, ICommandCallback<T> callback, CommandType commandType = CommandType.Text)
    {
        return Execute(new SimpleCommandCreator(sql, commandType), callback);
    }

    1 reference
    T Execute<T>(ICommandCreator creator, ICommandCallback<T> callback)...
}
```

Figure 13: Overide method Execute

Hàm **Execute** mới thêm vào chỉ cần khởi tạo lớp **SimpleCommandCreator** truyền vào 2 tham số là sql và commandType sau đó gọi hàm **Execute** xem *Figure 10* trước đó đã được cài đặt trước đây.

- ✓ Tạo interface **ICommandSetter**

Quay trở lại interface **ICommandCallback** làm thế nào mà có thể thực hiện truy vấn 3 trường hợp là **ExecuteNonQuery**, **ExecuteReader**, **ExecuteScalar**.

Bước tiếp theo ta sẽ cụ thể hóa trường hợp ExecuteReader cách thực hiện như sau

Tạo interface ICommandSetter

```
0 references
public interface ICommandSetter
{
    0 references
    void SetValues(IDbCommand command);
}
```

Figure 14: ICommandSetter

Mục đích của việc tạo interface này là vì trong các câu truy vấn thường có các tham số truyền vào tham số, ví dụ trường hợp lấy ra **Role** theo **RoleId** lúc này ta phải tham số xuống là @Id, trường hợp này ta sẽ được đảm nhận bằng 1 interface **ICommandSetter**

- ✓ Interface **IDataReaderExtractor**

Việc thực thi **ExecuteReader** ta sẽ phó thác cho interface IDataReaderExtractor nhận nhiệm vụ này với 1 hàm duy nhất là **ExtractData**

```
0 references
public interface IDataReaderExtractor<T>
{
    0 references
    T ExtractData(IDataReader reader);
}
```

Figure 15: IDataReaderExtractor

Đây là interface đảm nhận thực thi ExecuteReader bằng phương thức ExtractData, lưu ý rằng phương thức này trả về có thể là 1 đối tượng, có thể trả về 1 List object hoặc có thể trả về 1 Dictionary tùy thuộc người dùng mong muốn.

- ✓ Class **CommandReaderCallback**

Quay trở lại interface **ICommandCallback** ta sẽ cụ thể hóa interface này cho trường hợp ExecuteReader bằng class **CommandReaderCallback**

```
1 reference
public class CommandReaderCallback<T> : ICommandCallback<T>
{
    ICommandSetter setter;
    IDataReaderExtractor<T> extractor;
    1 reference
    public CommandReaderCallback(ICommandSetter setter, IDataReaderExtractor<T> extractor)
    {
        this.setter = setter;
        this.extractor = extractor;
    }
    2 references
    public T Handle(IDbCommand command)
    {
        IDataReader reader = null;
        try
        {
            if (setter != null)
            {
                setter.SetValues(command);
            }
            reader = command.ExecuteReader();
            return extractor.ExtractData(reader);
        }
        finally
        {
            if (reader != null)
            {
                reader.Dispose();
            }
        }
    }
}
```

Figure 16: CommandReaderCallback

Trong lớp **CommandReaderCallback** constructor được truyền vào 2 interface là **ICommandSetter** và **IDataReaderExtractor** và 1 method Handle nhận tham số là **IDbCommand**. Một lưu ý là vì trong quá trình truy vấn sẽ có trường hợp không có tham số nên interface **ICommandSetter** có thể có trường hợp null nên cần được kiểm tra nếu khác null thì mới tiến hành set tham số truyền vào.

- ✓ Phương thức Query trong interface IDbContext

Để thân thiện hơn với người dùng, tao vào lại Interface **IDbContext** và thêm vào 1 method là Query, phương thức này chỉ đảm nhận việc đọc dữ liệu tức là nó chỉ thực thi **ExecuteReader**

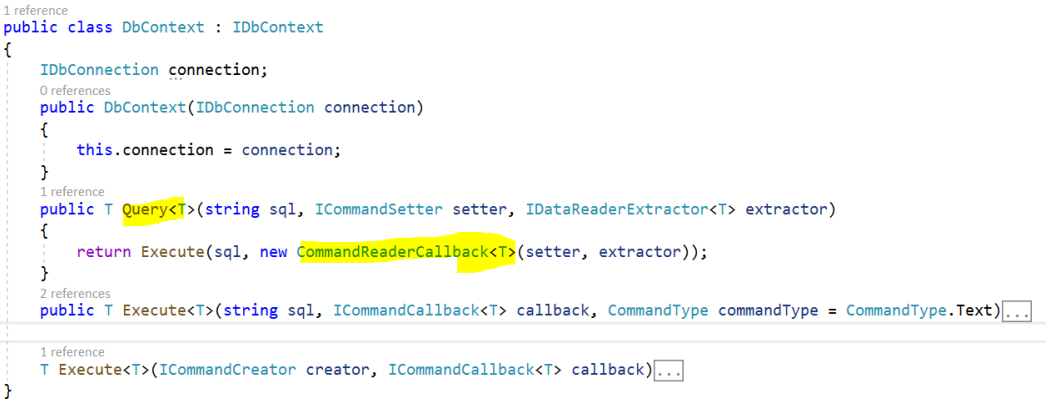
```
1 reference
public interface IDbContext
{
    1 reference
    T Execute<T>(string sql, ICommandCallback<T> callback, CommandType commandType = CommandType.Text);
    0 references
    T Query<T>(string sql, ICommandSetter setter, IDataReaderExtractor<T> extractor);
}
```

Figure 17: Phương thức Query

Hàm Query sẽ được tryền vào 2 tham số quan trọng là **ICommandSetter** để thiết lập tham số cho truy vấn và **IDataReaderExtractor** thực hiện đọc dữ liệu từ **ExecuteReader**

- ✓ Overide phương thức Query trong class DbContext

Sau khi thêm phương thức Query vào interface **IDbContext** ta buộc phải vào Class **DbContext** cũng sẽ cài đặt thêm phương thứ Query



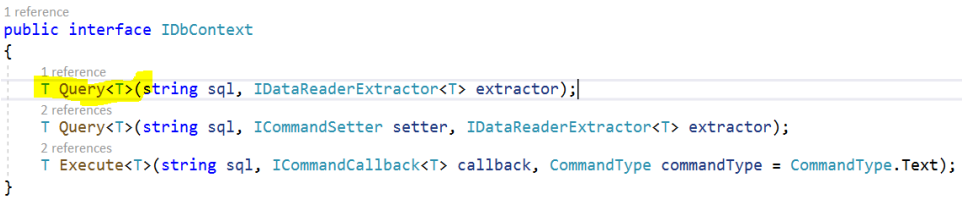
```
1 reference
public class DbContext : IDbContext
{
    IDbConnection connection;
    0 references
    public DbContext(IDbConnection connection)
    {
        this.connection = connection;
    }
    1 reference
    public T Query<T>(string sql, ICommandSetter setter, IDataReaderExtractor<T> extractor)
    {
        return Execute(sql, new CommandReaderCallback<T>(setter, extractor));
    }
    2 references
    public T Execute<T>(string sql, ICommandCallback<T> callback, CommandType commandType = CommandType.Text)...

    1 reference
    T Execute<T>(ICommandCreator creator, ICommandCallback<T> callback)...
}
```

Figure 18: Overidate method Query

Trong phương thức **Query** ta khởi tạo lớp **CommandReaderCallback** truyền vào 2 tham số **ICommandSetter** và **IDataReaderExtractor**. Sau đó gọi hàm Execute được thực hiện trước đó xem figure 13 truyền vào câu sql và đối tượng của lớp **CommandReaderCallback**.

- ✓ Chồng hàm Phương thức **Query** trong interface **IDbContext**

Trong interface **IDbContext** ta thêm 1 phương thức Query với 2 tham số là sql và extractor đây là phương thức thực hiện trong trường hợp câu truy vấn không sử dụng tham số



```
1 reference
public interface IDbContext
{
    1 reference
    T Query<T>(string sql, IDataReaderExtractor<T> extractor);
    2 references
    T Query<T>(string sql, ICommandSetter setter, IDataReaderExtractor<T> extractor);
    2 references
    T Execute<T>(string sql, ICommandCallback<T> callback, CommandType commandType = CommandType.Text);
}
```

Figure 19: Method Query

Trong class **DbContext** ta cũng thêm hàm Query hàm này sẽ gọi lại hàm **Query** trước đó trong trường hợp **ICommandSetter** là giá trị null

```
1 reference
public class DbContext : IDbContext
{
    IDbConnection connection;
    0 references
    public DbContext(IDbConnection connection)...
    1 reference
    public T Query<T>(string sql, IDataReaderExtractor<T> extractor)
    {
        return Query(sql, null, extractor);
    }
    2 references
    public T Query<T>(string sql, ICommandSetter setter, IDataReaderExtractor<T> extractor)...

    2 references
    public T Execute<T>(string sql, ICommandCallback<T> callback, CommandType commandType = CommandType.Text)...

    1 reference
    T Execute<T>(ICommandCreator creator, ICommandCallback<T> callback)...


}
```

Figure 20: Overidate method Query

- ✓ Interface IRowMapper

Quay lại interface **IDataReaderExtractor** vẫn chỉ là interface tổng quát để có thể chi tiết hơn ta tạo tiếp 1 interface **IRowMapper**

```
0 references
public interface IRowMapper<T>
{
    0 references
    T MapRow(IDataReader reader);
}
```

Figure 21: interface IRowMapper

Lưu ý rằng Interface **IRowMapper** nó chỉ đảm nhận việc đọc từng dòng dữ liệu trong 1 bảng bất kỳ điều này cho thấy interface này tỏ ra rất hữu ích việc đọc dữ liệu cụ thể như thế nào ta chỉ cần cài đặt lại interfae này là có thể đọc được mọi bảng dữ liệu.

- ✓ Class **ListDataReaderExtractor**

Xét trường hợp đọc dữ liệu trả về nhiều dòng dữ liệu (trường hợp trả về một danh sách) ta cài đặt class **ListDataReaderExtractor** cụ thể cho interface **IDataReaderExtractor**

```
1 reference
public class ListDataReaderExtractor<T> : IDataReaderExtractor<List<T>>
{
    IRowMapper<T> mapper;
    0 references
    public ListDataReaderExtractor(IRowMapper<T> mapper)
    {
        this.mapper = mapper;
    }
    2 references
    public List<T> ExtractData(IDataReader reader)
    {
        List<T> list = new List<T>();
        while (reader.Read())
        {
            list.Add(mapper.MapRow(reader));
        }
        return list;
    }
}
```

Figure 22: class ListDataReaderExtractor

Trong lớp **ListDataReaderExtractor** truyền vào constructor là một interface **IRowMapper** để có thể đọc bất kỳ một bảng dữ liệu nào. Lúc này trong phương thức **ExtractData** với tham số truyền vào là **IDataReader** lúc này ta khởi tạo một List dạng Generic sau đó lặp từng dòng và sử dụng đối tượng tham chiếu **IRowMapper** và gọi hàm **MapRow** là xong, cụ thể gán dữ liệu như thế nào thỉ chỉ class Implement lại Interface **IRowMapper** thì quyết định điều này.

- ✓ Overload phương thức Query trong interface IDbContext

Quay lại interface **IDbContext** ta thêm 1 phương thức **Query**

```
1 reference
public interface IDbContext
{
    1 reference
    List<T> Query<T>(string sql, IRowMapper<T> mapper, CommandType commandType = CommandType.Text);
    2 references
    T Query<T>(string sql, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text);
    2 references
    T Query<T>(string sql, ICommandSetter setter, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text);
    2 references
    T Execute<T>(string sql, ICommandCallback<T> callback, CommandType commandType = CommandType.Text);
}
```

Figure 23: Overload method Query

Lưu ý rằng phương thức này có giá trị trả về là một List khi thực hiện gọi phương thức này người dùng chỉ cần implement lại interface **IRowMapper** là truyền câu truy vấn tương ứng trên 1 bảng bất kỳ là có thể lấy được danh sách các dòng tương ứng trên bảng

Tương ứng ta thực hiện cài đặt thêm phương thức **Query** trên class **DbContext**

```
1 reference
public class DbContext : IDbContext
{
    IDbConnection connection;
    0 references
    public DbContext(IDbConnection connection)...
    1 reference
    public List<T> Query<T>(string sql, IRowMapper<T> mapper, CommandType commandType = CommandType.Text)
    {
        return Query(sql, new ListDataReaderExtractor<T>(mapper), commandType);
    }
    2 references
    public T Query<T>(string sql, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text)...
    2 references
    public T Query<T>(string sql, ICommandSetter setter, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text)...

    2 references
    public T Execute<T>(string sql, ICommandCallback<T> callback, CommandType commandType = CommandType.Text)...

    1 reference
    T Execute<T>(ICommandCreator creator, ICommandCallback<T> callback)...
}
```

Figure 24: Overide method Query

Phương thức **Query** này được cài đặt bằng cách gọi phương thức **Query** đã được cài đặt trước đó xem Figure 20, tham số quan trọng được truyền vào là khởi tạo lớp **ListDataReaderExtractor**

- ✓ Class **RowMapper**

Để thực hiện đọc dữ liệu trên bảng Role ta cài đặt lớp **RoleMapper** implement cho interface **IRowMapper**

```
0 references
class RoleMapper : IRowMapper<Role>
{
    2 references
    public Role MapRow(IDataReader reader)
    {
        return new Role
        {
            Id = (Guid)reader["RoleId"],
            Name = (string)reader["RoleName"]
        };
    }
}
```

- ✓ Phương thức **GetRoles**

Lúc này quay trở lại class **RoleRepository** ta dễ dàng thực hiện phương thức **GetRoles** cực kỳ đơn giản

Trước tiên ta tạo abstract class **BaseRepository** với mục đích tạo lớp dùng chung một thuộc tính **dbContext** cho nhiều class Repository khác nhau.

```
1 reference
public abstract class BaseRepository
{
    protected IDbContext dbContext;
    0 references
    public BaseRepository(IDbContext dbContext)
    {
        this.dbContext = dbContext;
    }
}
```

Figure 25:class BaseRepository

Lúc này class **RoleRepository** được cập nhật lại như sau

```
1 reference
public class RoleRepository : BaseRepository
{
    0 references
    public RoleRepository(IDbContext dbContext):base(dbContext)
    {
    }
    0 references
    public List<Role> GetRoles()
    {
        return dbContext.Query("SELECT * FROM Role", new RoleMapper());
    }
    /* readonly static string connectionString = "Data Source=.;Initial
}
```

Trong phương thức **GetRoles** ta sử dụng đối tượng **dbContext** rồi gọi hàm **Query** truyền vào sql và khởi tạo lớp **RoleMapper** là có thể đọc được dữ liệu trên bảng Role. Việc này có thể thực hiện dễ dàng trên bảng bất kỳ nào khác.

- ✓ Class **SingeDataReaderExtractor**

Quay trở lại interface **IDataReaderExtractor** làm thế nào có thể xử lý trường hợp đọc 1 dòng dữ liệu điều này đơn giản thực hiện bằng cách tạo 1 class **SingeDataReaderExtractor**. Các bạn chú ý xem qua hàm xử lý cực kỳ đơn giản như trong hình sau:

```
1 reference
public class SingleDataReaderExtractor<T> : IDataReaderExtractor<T>
{
    IRowMapper<T> mapper;
    0 references
    public SingleDataReaderExtractor(IRowMapper<T> mapper)
    {
        this.mapper = mapper;
    }
    3 references
    public T ExtractData(IDataReader reader)
    {
        if (reader.Read())
        {
            return mapper.MapRow(reader);
        }
        return default(T);
    }
}
```

Figure 26: class SingleDataReaderExtractor

Lúc này trong phương thức ExtractData ta chỉ sử dụng một câu lệnh if rồi sau đó sử dụng đối tượng IRowMapper và gọi hàm MapRow truyền vào đối tượng reader là đã xử lý đọc 1 dòng.

- ✓ Phương thức **QueryOne**

Quay lại interface ta thêm 1 phương thức **QueryOne**

```
4 references
public interface IDbContext
{
    1 reference
    T QueryOne<T>(string sql, ICommandSetter setter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text);
    2 references
    List<T> Query<T>(string sql, IRowMapper<T> mapper, CommandType commandType = CommandType.Text);
    2 references
    T Query<T>(string sql, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text);
    3 references
    T Query<T>(string sql, ICommandSetter setter, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text);
    2 references
    T Execute<T>(string sql, ICommandCallback<T> callback, CommandType commandType = CommandType.Text);
}
```

Figure 27: Method QueryOne

Trong phương thức **QueryOne** lúc này chỉ trả về 1 đối tượng nên không sử dụng List và thường trong phương thức này sẽ phải truyền vào tham số nên lúc này đương nhiên phải có interface **ICommandSetter**

Tương tự ta cũng phải thêm vào class **DbContext** phương thức **QueryOne**

```
1 reference
public class DbContext : IDbContext
{
    IDbConnection connection;
    0 references
    public DbContext(IDbConnection connection)...
    1 reference
    public T QueryOne<T>(string sql, ICommandSetter setter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text)
    {
        return Query(sql, setter, new SingleDataReaderExtractor<T>(mapper), commandType);
    }
    2 references
    public List<T> Query<T>(string sql, IRowMapper<T> mapper, CommandType commandType = CommandType.Text)...
    2 references
    public T Query<T>(string sql, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text)...
    3 references
    public T Query<T>(string sql, ICommandSetter setter, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text)...

    2 references
    public T Execute<T>(string sql, ICommandCallback<T> callback, CommandType commandType = CommandType.Text)...

    1 reference
    T Execute<T>(ICommandCreator creator, ICommandCallback<T> callback)...
}
```

Figure 28: Implement method QueryOne

Trong phương thức QueryOne ta sẽ khởi tạo class **SingleDataReaderExtractor** được tạo ở *Figure 26* và lúc này ta cũng chỉ cần gọi lại hàm **Query** xem *Figure 18* là cho ra kết quả hoàn thiện của hàm **QueryOne**

- ✓ Class **Parameter**

Để cụ thể hóa interface **ICommandSetter** ta phải cài đặt lớp cụ thể để truyền vào **Parameter** đơn giản hơn. Ta thực hiện các bước sau:

Tạo class **Parameter**

```
0 references
public class Parameter
{
    0 references
    public string Name { get; set; }
    0 references
    public object Value { get; set; }
    0 references
    public DbType DbType { get; set; }
    0 references
    public ParameterDirection Direction { get; set; }
    0 references
    public IDataParameter DataParameter { get; set; }
}
```

Figure 29: class Parameter

Trong class này lưu ý rằng có 2 thuộc tính Direction và DataParameter nhằm mục đích lấy giá trị tham số output hoặc return trong thủ tục

- ✓ Interface **IDynamicParameter**

Để cho việc khởi tạo và quản lý Parameter đơn giản hơn, ta tiếp thục tạo thêm interface **IDynamicParameter**

```
0 references
public interface IDynamicParameter
{
    0 references
    void Add(string name, object value = null, DbType dbType = DbType.String, ParameterDirection direction = ParameterDirection.Input);
    0 references
    void Handle(IDbCommand command);
    0 references
    T Get<T>(string name);
}
```

Figure 30: interface IDynamicParameter

Trong interface này hàm **Add** có nhiệm vụ thêm các tham số vào câu truy vấn ta chỉ cần truyền vào tên tham số và giá trị của tham số. Phương thức **Handle** nhằm mục đích set tham số thực sự vào **Command**. Kèm thêm phương thức **Get** với mục đích lấy giá trị của tham số output và return của thủ tục nội tại.

- ✓ Class **SimpleCommandSetter**

Thực hiện cài đặt lớp cụ thể cho interface **ICommandSetter** bằng class **SimpleCommandSetter**

```
1 reference
public class SimpleCommandSetter : ICommandSetter
{
    IDynamicParameter parameter;
    0 references
    public SimpleCommandSetter(IDynamicParameter parameter)
    {
        this.parameter = parameter;
    }
    2 references
    public void SetValues(IDbCommand command)
    {
        parameter.Handle(command);
    }
}
```

Figure 31: class SimpleCommandSetter

- ✓ Phương thức **QueryOne** và **Query** với tham số **IDynamicParameter**

Quay trở lại interface **IDbContext** ta thêm 2 phương thức **QueryOne** và **Query**

```
4 references
public interface IDbContext
{
    0 references
    T QueryOne<T>(string sql, IDynamicParameter parameter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text);
    0 references
    List<T> Query<T>(string sql, IDynamicParameter parameter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text);
    1 reference
    T QueryOne<T>(string sql, ICommandSetter setter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text);
    2 references
    List<T> Query<T>(string sql, IRowMapper<T> mapper, CommandType commandType = CommandType.Text);
    2 references
    T Query<T>(string sql, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text);
    3 references
    T Query<T>(string sql, ICommandSetter setter, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text);
    2 references
    T Execute<T>(string sql, ICommandCallback<T> callback, CommandType commandType = CommandType.Text);
}
```

2 phương thức này đều có tham số là **IDynamicParameter** nhằm mục đích truyền tham số vào trong câu truy vấn.

Và ta cũng phải cài đặt 2 phương thức QueryOne và Query vào trong class DbContext

```
1 reference
public class DbContext : IDbContext
{
    IDbConnection connection;
    0 references
    public DbContext(IDbConnection connection)...
    1 reference
    public T QueryOne<T>(string sql, IDynamicParameter parameter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text)
    {
        return QueryOne(sql, new SimpleCommandSetter(parameter), mapper, commandType);
    }
    1 reference
    public List<T> Query<T>(string sql, IDynamicParameter parameter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text)
    {
        return Query(sql, new SimpleCommandSetter(parameter), new ListDataReaderExtractor<T>(mapper), commandType);
    }
    2 references
    public T QueryOne<T>(string sql, ICommandSetter setter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text)...
    2 references
    public List<T> Query<T>(string sql, IRowMapper<T> mapper, CommandType commandType = CommandType.Text)...
    2 references
    public T Query<T>(string sql, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text)...
    4 references
    public T Query<T>(string sql, ICommandSetter setter, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text)...
    2 references
    public T Execute<T>(string sql, ICommandCallback<T> callback, CommandType commandType = CommandType.Text)...
    1 reference
    T Execute<T>(ICommandCreator creator, ICommandCallback<T> callback)...
}
```

Figure 32: imlement phương thức QueryOne và Query

Trong phương thức **QueryOne** ta thực hiện cài đặt bằng cách gọi phương thức **QueryOne** đã được thực hiện trước đó trong Figure 28 với tham số truyền vào là khởi tạo lớp cụ thể **SimpleCommanSetter**. Phương thức **Query** cũng được thực hiện tương tự.

- ✓ Class **DynamicParameter**

Để có thể thực hiện truyền tham số và câu truy vấn ta thực hiện tạo class **DynamicParameter** để hiện thực hóa cho interface **IDynamicParameter**

```
1 reference
public class DynamicParameter : IDynamicParameter
{
    IDictionary<string, Parameter> paramters;
    0 references
    public DynamicParameter(int capacicy)
    {
        paramters = capacicy > 0 ? new Dictionary<string, Parameter>(capacicy) : new Dictionary<string, Parameter>();
    }
    1 reference
    public void Add(string name, object value = null, DbType dbType = DbType.String, ParameterDirection direction = ParameterDirection.Input)
    {
        paramters[name] = new Parameter{ Name = name, Value = value, DbType = dbType, Direction = direction };
    }
    1 reference
    public T Get<T>(string name)
    {
        return (T)paramters[name].DataParameter.Value;
    }
    2 references
    public void Handle(IDbCommand command)
    {
        foreach (Parameter item in paramters.Values)
        {
            IDataParameter dataParameter = command.CreateParameter();
            dataParameter.ParameterName = item.Name;
            dataParameter.Value = item.Value;
            dataParameter.Direction = item.Direction;
            dataParameter.DbType = item.DbType;
            item.DataParameter = dataParameter;
            command.Parameters.Add(dataParameter);
        }
    }
}
```

Figure 33: class DynamicParameter

Trong class **DynamicParameter** này xử lý tất cả các vấn đề liên quan đến Parameter truyền vào hoặc là lấy tham số đầu ra hoặc giá trị trả về của thủ tục. Để công việc tìm kiếm xử lý nhanh ta sẽ sử dụng một Dictionary với khóa là kiểu string và giá trị là một class **Parameter**.

- ✓ Phương thức **GetRole**

Quay trở lại lớp **RoleRepository** ta thực hiện cài đặt phương thức **GetRole** như sau

```
1 reference
public class RoleRepository : BaseRepository
{
    0 references
    public RoleRepository(IDbContext dbContext)...
    0 references
    public List<Role> GetRoles()...
    0 references
    public Role GetRole(int id)
    {
        IDynamicParameter parameter = new DynamicParameter(1);
        parameter.Add("@Id", id, DbType.Guid);
        return dbContext.QueryOne("SELECT * FROM Role WHERE RoleId = @Id", parameter, new RoleMapper());
    }
    /* readonly static string connectionString = "Data Source=.;Initial Catalog=BikeStore;Integrated Security=True"; ...
}
```

Figure 34: method GetRole

Trong phương thức **GetRole** ta khởi tạo class **DynamicParameter** rồi truyền vào tham số id sau đó sử dụng đối tượng dbContext rồi gọi hàm **QueryOne** truyền vào đầy đủ tham số là ta phương thức **GetRole** hoàn chỉnh.

- ✓ Class **CommandUpdateCallback**

Quay trở lại interface **ICommandCallback** ta muốn cụ thể hóa trường hợp **ExecuteNoneQuery** đươc thực hiện bằng cách cài đặ class **CommandUpdateCallback**

```
1 reference
public class CommandUpdateCallback : ICommandCallback<int>
{
    ICommandSetter setter;
    0 references
    public CommandUpdateCallback(ICommandSetter setter)
    {
        this.setter = setter;
    }
    3 references
    public int Handle(IDbCommand command)
    {
        try
        {
            if (setter != null)
            {
                setter.SetValues(command);
            }
            return command.ExecuteNonQuery();
        }catch(Exception ex)
        {
            throw new Exception("Handle Update" + ex.Message);
        }
    }
}
```

Figure 35: class CommandUpdateCallback

- ✓ Phương thức **Update** trong interface **IDbContext**

Trong interface **IDbContext** ta thêm hàm **Update** với 2 tham số quan trong là sql và **ICommandSetter**

```
4 references
public interface IDbContext
{
    0 references
    int Update(string sql, ICommandSetter setter, CommandType commandType = CommandType.Text);
    2 references
    T QueryOne<T>(string sql, IDynamicParameter parameter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text);
    1 reference
    List<T> Query<T>(string sql, IDynamicParameter parameter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text);
    2 references
    T QueryOne<T>(string sql, ICommandSetter setter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text);
    2 references
    List<T> Query<T>(string sql, IRowMapper<T> mapper, CommandType commandType = CommandType.Text);
    2 references
    T Query<T>(string sql, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text);
    4 references
    T Query<T>(string sql, ICommandSetter setter, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text);
    2 references
    T Execute<T>(string sql, ICommandCallback<T> callback, CommandType commandType = CommandType.Text);
}
```

Figure 36: method Update

Đừng quên tạo thêm hàm Update vào trong class DbContext

```
1 reference
public class DbContext : IDbContext
{
    IDbConnection connection;
    0 references
    public DbContext(IDbConnection connection)...
    1 reference
    public int Update(string sql, ICommandSetter setter, CommandType commandType = CommandType.Text)
    {
        return Execute(sql, new CommandUpdateCallback(setter), commandType);
    }
    2 references
    public T QueryOne<T>(string sql, IDynamicParameter parameter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text)...
    1 reference
    public List<T> Query<T>(string sql, IDynamicParameter parameter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text)...
    2 references
    public T QueryOne<T>(string sql, ICommandSetter setter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text)...
    2 references
    public List<T> Query<T>(string sql, IRowMapper<T> mapper, CommandType commandType = CommandType.Text)...
    2 references
    public T Query<T>(string sql, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text)...
    4 references
    public T Query<T>(string sql, ICommandSetter setter, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text)...
    3 references
    public T Execute<T>(string sql, ICommandCallback<T> callback, CommandType commandType = CommandType.Text)...
    1 reference
    T Execute<T>(ICommandCreator creator, ICommandCallback<T> callback)...
}
```

Figure 37: implement Update

Thật khá dễ dàng thêm vào class **DbContext** phương thức **Update**, trong phương thức này ta chỉ cần khởi tạo class **CommandUpdateCallback** vào gọi hàm Execute đã được cài đặt trước đó xem *Figure 13*.

- ✓ Phương thức **Update** với interface **IDynamicParameter**

Để người dùng sử dụng dễ dàng hơn ta thêm một phương thức **Update** với tham số truyền vào là một **IDynamicParameter**

```
4 references
public interface IDbContext
{
    1 reference
    int Update(string sql, IDynamicParameter parameter, CommandType commandType = CommandType.Text);
    2 references
    int Update(string sql, ICommandSetter setter, CommandType commandType = CommandType.Text);
    2 references
    T QueryOne<T>(string sql, IDynamicParameter parameter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text);
    1 reference
    List<T> Query<T>(string sql, IDynamicParameter parameter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text);
    2 references
    T QueryOne<T>(string sql, ICommandSetter setter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text);
    2 references
    List<T> Query<T>(string sql, IRowMapper<T> mapper, CommandType commandType = CommandType.Text);
    2 references
    T Query<T>(string sql, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text);
    4 references
    T Query<T>(string sql, ICommandSetter setter, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text);
    3 references
    T Execute<T>(string sql, ICommandCallback<T> callback, CommandType commandType = CommandType.Text);
}
```

Figure 38: method update với interface IDynamicParameter

Và cũng phải cài đặt lại phương thức Update trong class **DbContext**.

```
1 reference
public class DbContext : IDbContext
{
    IDbConnection connection;
    0 references
    public DbContext(IDbConnection connection)...
    1 reference
    public int Update(string sql, IDynamicParameter parameter, CommandType commandType = CommandType.Text)
    {
        return Update(sql, new SimpleCommandSetter(parameter), commandType);
    }
    2 references
    public int Update(string sql, ICommandSetter setter, CommandType commandType = CommandType.Text)...
    2 references
    public T QueryOne<T>(string sql, IDynamicParameter parameter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text)...
    1 reference
    public List<T> Query<T>(string sql, IDynamicParameter parameter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text)...
    2 references
    public T QueryOne<T>(string sql, ICommandSetter setter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text)...
    2 references
    public List<T> Query<T>(string sql, IRowMapper<T> mapper, CommandType commandType = CommandType.Text)...
    2 references
    public T Query<T>(string sql, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text)...
    4 references
    public T Query<T>(string sql, ICommandSetter setter, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text)...
    3 references
    public T Execute<T>(string sql, ICommandCallback<T> callback, CommandType commandType = CommandType.Text)...
    1 reference
    T Execute<T>(ICommandCreator creator, ICommandCallback<T> callback)...
}
```

Figure 39: implement phương thức Update

Phương thức **Update** mới được thêm vào được cài đặt bằng cách khởi tạo **SimpleCommandSetter** sau đó gọi hàm Update đã được viết trước đó xem *Figure 37*.

- ✓ Phương thức **Add**

Quay trở lại lớp **RoleRepository** ta dễ dàng thêm phương thức **Add** như sau

```
1 reference
public class RoleRepository : BaseRepository
{
    0 references
    public RoleRepository(IDbContext dbContext)...
    0 references
    public List<Role> GetRoles()...
    0 references
    public Role GetRole(int id)...
    0 references
    public int Add(Role obj)
    {
        IDynamicParameter parameter = new DynamicParameter(1);
        parameter.Add("@Name", obj.Name);
        return dbContext.Update("INSERT INTO Role (RoleName) VALUES (@Name)", parameter);
    }
    /* readonly static string connectionString = "Data Source=.;Initial Catalog=BikeStore;
}
```

Figure 40: phương thức Add

Việc thực hiện **Insert**, **Update**, **Delete** đều có thể sử dụng phương thức **Update** của class **DbContext** ta chỉ cần gọi hàm update và truyền 2 tham số là sql và **DynamicParameter**.

- ✓ Class **CommandScalarCallback**

Quay trở lại interface **ICommandCallback** ta muốn thực hiện truy vấn **ExecuteScalar** ta hiện thực hóa trường hợp này bằng một lớp cụ thể là class **CommandScalarCallback**

```
1 reference
public class CommandScalarCallback<T> : ICommandCallback<T>
{
    ICommandSetter setter;
    1 reference
    public CommandScalarCallback(ICommandSetter setter)
    {
        this.setter = setter;
    }
    4 references
    public T Handle(IDbCommand command)
    {
        try
        {
            if (setter != null)
            {
                setter.SetValues(command);
            }
            return (T)command.ExecuteScalar();
        }
        catch(Exception ex)
        {
            throw new Exception("Handle Scalar " + ex.Message);
        }
    }
}
```

Figure 41: class CommandScalarCallback

Trong class **CommandScalarCallback** ta sử dụng thực thi **ExecuteScalare** chỉ trả về 1 giá trị đơn tùy thuộc vào kiểu giá trị trả về do người dùng quyết định.

- ✓ Phương thức **Scalar**

Cập nhật interface **IDdContext** như sau

```
5 references
public interface IDbContext
{
    1 reference
    T Scalar<T>(string sql, IDynamicParameter parameter, CommandType commandType = CommandType.Text);
    3 references
    T Scalar<T>(string sql, ICommandSetter setter = null, CommandType commandType = CommandType.Text);
    2 references
    int Update(string sql, IDynamicParameter parameter, CommandType commandType = CommandType.Text);
    2 references
    int Update(string sql, ICommandSetter setter, CommandType commandType = CommandType.Text);
    2 references
    T QueryOne<T>(string sql, IDynamicParameter parameter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text);
    1 reference
    List<T> Query<T>(string sql, IDynamicParameter parameter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text);
    2 references
    T QueryOne<T>(string sql, ICommandSetter setter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text);
    2 references
    List<T> Query<T>(string sql, IRowMapper<T> mapper, CommandType commandType = CommandType.Text);
    2 references
    T Query<T>(string sql, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text);
    4 references
    T Query<T>(string sql, ICommandSetter setter, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text);
    4 references
    T Execute<T>(string sql, ICommandCallback<T> callback, CommandType commandType = CommandType.Text);
}
```

Figure 42: phương thức Scalar

Trong interface IDbContext ta cập nhật thêm 2 phương thức **Scalar** tương ứng với trường hợp sử dụng **ICommandSetter** và trường hợp sử dụng **IDynamicParameter**. Lưu ý ở đây phương thức sử dụng **ICommandSetter** có giá trị mặc định là null để chỉ định khi không cần truyền tham số vào câu truy vấn SQL.

Tương ứng trong class **DbContext** ta cũng phải thêm 2 phương thức **Scalar**

```
1 reference
public class DbContext : IDbContext
{
    IDbConnection connection;
    1 reference
    public DbContext(IDbConnection connection)...
    1 reference
    public T Scalar<T>(string sql, IDynamicParameter parameter, CommandType commandType = CommandType.Text)
    {
        return Scalar<T>(sql, new SimpleCommandSetter(parameter), commandType);
    }
    3 references
    public T Scalar<T>(string sql, ICommandSetter setter = null, CommandType commandType = CommandType.Text)
    {
        return Execute(sql, new CommandScalarCallback<T>(setter), commandType);
    }
            2 references
            public int Update(string sql, IDynamicParameter parameter, CommandType commandType = CommandType.Text)...
    2 references
    public int Update(string sql, ICommandSetter setter, CommandType commandType = CommandType.Text)...
    2 references
    public T QueryOne<T>(string sql, IDynamicParameter parameter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text)...
    1 reference
    public List<T> Query<T>(string sql, IDynamicParameter parameter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text)...
    2 references
    public T QueryOne<T>(string sql, ICommandSetter setter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text)...
    2 references
    public List<T> Query<T>(string sql, IRowMapper<T> mapper, CommandType commandType = CommandType.Text)...
    2 references
    public T Query<T>(string sql, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text)...
    4 references
    public T Query<T>(string sql, ICommandSetter setter, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text)...
    4 references
    public T Execute<T>(string sql, ICommandCallback<T> callback, CommandType commandType = CommandType.Text)...
    1 reference
    T Execute<T>(ICommandCreator creator, ICommandCallback<T> callback)...
}
```

Figure 43: implement Scalar

Lưu ý trong phương thức Scalar với tham số là ICommandSetter ta phải khởi tạo class **CommandScalarCallback** được cài đặt trước đó, lúc này chỉ cần gọi hàm **Execute** là có được kết quả thực thi tương ứng.

- ✓ Phương thức Count

Trong class **RoleRepository** ta muốn thực hiện truy vấn đếm xem có bao nhiêu dòng trong bảng **Role** ta thực hiện thông qua phương thức **Count** như sau:

```
3 references
public class RoleRepository : BaseRepository
{
    1 reference
    public RoleRepository(IDbContext dbContext)...
    1 reference
    public List<Role> GetRoles()...
    0 references
    public Role GetRole(int id)...
    1 reference
    public int Add(Role obj)...
    1 reference
    public int Count()
    {
        return dbContext.Scalar<int>("SELECT COUNT(*) FROM Role");
    }
    /* readonly static string connectionString = "Data Source=.;In
}
```

Figure 44: phương thức Count

Trong phương thức **Count** ta sử dụng đối tượng dbContext và gọi đến phương thức **Scalar** rồi truyền câu truy vấn thống kê vào là ta có thể lấy được kết quả trả về.

- ✓ Interface **IQueryMultiple**

Quay lại interface **ICommandCallback** ta tìm hiểu thêm làm thế nào để thực hiện truy vấn trên nhiều bảng.

Ta tạo interface **IQueryMultiple**

```
8 references
public interface IQueryMultiple : IDisposable
{
    3 references
    T Query<T>(IDataReaderExtractor<T> extractor);
    2 references
    List<T> Query<T>(IRowMapper<T> mapper);
    2 references
    T QueryOne<T>(IRowMapper<T> mapper);
}
```

Figure 45: interface IQueryMultiple

Trong interface **IQueryMultiple** ta viết 3 phương thức đối với phương thức **Query** với tham số là **IDataReaderExtractor** nhằm xử lý trường hợp có thể trả về bất kỳ một kiểu dữ liệu nào ví dụ có thể là 1 Dictionary. 2 trường hợp còn lại là trường hợp cụ thể trả về 1 đối tượng hoặc một List đối tượng.

- ✓ Class **QueryMultiple**

Tạo class **QueryMultiple** để implement cho interface **IQueryMultiple**.

```
public class QueryMultiple : IQueryMultiple
{
    IDataReader reader; bool isFirst = true;
    1 reference
    public QueryMultiple(IDataReader reader)
    {
        this.reader = reader;
    }
    1 reference
    void TheFirst()
    {
        if (isFirst) { isFirst = false; } else { reader.NextResult(); }
    }
    3 references
    public T Query<T>(IDataReaderExtractor<T> extractor)
    {
        TheFirst();
        return extractor.ExtractData(reader);
    }
    2 references
    public List<T> Query<T>(IRowMapper<T> mapper)
    {
        return Query(new ListDataReaderExtractor<T>(mapper));
    }
    2 references
    public T QueryOne<T>(IRowMapper<T> mapper)
    {
        return Query(new SingleDataReaderExtractor<T>(mapper));
    }
    0 references
    public void Dispose()
    {
        if (reader != null) { reader.Dispose(); }
    }
}
```

Figure 46: class QueryMultiple

Trong phương thức Query ta khởi tạo class **ListDataReaderExtractor** để xử lý trường hợp trả về 1 List object. Tương tự trong phương thức **QueryOne** ta cũng khởi tạo class **SingleDataReaderExtractor** để trả về 1 object duy nhất.

- ✓ Class **CommandMultipleQueryCallback**

Tiếp tục tạo lớp **CommandMultipleQueryCallback** implement interface **ICommandCallback**

```
1 reference
public class CommandQueryMultipleCallback : ICommandCallback<IQueryMultiple>
{
    ICommandSetter setter;
    1 reference
    public CommandQueryMultipleCallback(ICommandSetter setter)
    {
        this.setter = setter;
    }
    5 references
    public IQueryMultiple Handle(IDbCommand command)
    {
        try
        {
            if (setter != null)
            {
                setter.SetValues(command);
            }
            return new QueryMultiple(command.ExecuteReader());

        }catch(Exception ex)
        {
            throw new Exception(ex.Message);
        }
    }
}
```

Figure 47: class CommandMultipleQueryCallback

Trong phương thức **Handle** ta sẽ trả về 1 Interface **IQueryMultiple** để người dùng có thể tùy ý duyệt theo từng table cụ thể.

- ✓ Method **QueryMultiple**

Lúc này ta vào interface **IDbContext** ta cập nhật thêm 2 phương thức QueryMultiple

```
6 references
public interface IDbContext
{
    2 references
    IQueryMultiple QueryMultiple(string sql, IDynamicParameter parameter, CommandType commandType = CommandType.Text);
    2 references
    IQueryMultiple QueryMultiple(string sql, ICommandSetter setter = null, CommandType commandType = CommandType.Text);
    1 reference
    T Scalar<T>(string sql, IDynamicParameter parameter, CommandType commandType = CommandType.Text);
    3 references
    T Scalar<T>(string sql, ICommandSetter setter = null, CommandType commandType = CommandType.Text);
    2 references
    int Update(string sql, IDynamicParameter parameter, CommandType commandType = CommandType.Text);
    2 references
    int Update(string sql, ICommandSetter setter, CommandType commandType = CommandType.Text);
    2 references
    T QueryOne<T>(string sql, IDynamicParameter parameter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text);
    1 reference
    List<T> Query<T>(string sql, IDynamicParameter parameter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text);
    2 references
    T QueryOne<T>(string sql, ICommandSetter setter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text);
    3 references
    List<T> Query<T>(string sql, IRowMapper<T> mapper, CommandType commandType = CommandType.Text);
    2 references
    T Query<T>(string sql, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text);
    4 references
    T Query<T>(string sql, ICommandSetter setter, IDataReaderExtractor<T> extractor, CommandType commandType = CommandType.Text);
    5 references
    T Execute<T>(string sql, ICommandCallback<T> callback, CommandType commandType = CommandType.Text);
}
```

Figure 48: Method QueryMultiple

Hàm **QueryMultile** đơn giản truyền vào câu truy vấn và tham số nó sẽ trả về 1 interface **IQueryMultiple** để người dùng có thể sử dụng lấy dữ liệu từ các bảng khác nhau.

Và cũng không quên thêm vào 2 phương thức **QueryMultiple** trong class **DbContext**

Figure 49: implement method QueryMultiple

Trong phương thức QueryMultiple ta phải khởi tạo class **CommandQueryMultipleCallback** rồi gọi hàm **Execute** đã được cài đặt trước đó là ta có được 1 phương thức **QueryMultiple** hoàn hảo.

- ✓ Class **AccountRepository**

Thường khi truy vấn trên nhiều bảng ta sẽ tạo trước 1 procedure trong sql

```
CREATE PROC GetAccountRoles(@Id UNIQUEIDENTIFIER)
AS
BEGIN
    SELECT * FROM Account WHERE AccountId = @Id;
    SELECT Role.*, CAST(IIF(AccountId IS NULL, 0, 1)  AS BIT) AS Checked
        FROM AccountInRole RIGHT JOIN Role ON AccountId = @Id AND AccountInRole.RoleId = Role.RoleId;
END
```

Ta thấy rằng trong thủ tục trên bao gồm 2 câu truy vấn. Câu truy vấn thứ nhất chỉ trả về 1 dòng dữ liệu, và câu truy vấn thứ 2 trả về nhiều dòng dữ liệu.

Lúc này trong class **AccountRepository** ta viết phương thức **GetAcountRoles** như sau.

Figure 50: method GetAccountRoles

Trong phương thức **GetAccountRoles** ta sử dụng phương thức **QueryMultiple**, nó sẽ trả về 1 interface là IQueryMultiple. Từ đó ta sẽ đọc dữ liệu trên từng bảng bằng 2 phương thức **Query** và **QueryOne** trong interface IQueryMultiple rất dễ dàng.

Thực hiện BatchUpdate.

- ✓ Interface **IBatchUpdateCommandSetter**

Trước tiên ta tạo 1 interface **IBatchUpdateCommandSetter**

```
5 references
public interface IBatchUpdateCommandSetter
{
    2 references
    int BatchSize { get; }
    2 references
    void SetValues(IDynamicParameter parameter, int i);
}
```

Figure 51: interface IBatchUpdateCommandSetter

Thuộc tính BatchSize để biết rằng ta sẽ lập vòng for **BatchSize** lần để thực hiện **ExecuteNonQuery**. Phương thức **SetValues** nhằm mục tính thiết lập giá trị cho từng tham số tương ứng với phần tử thứ i trong 1 danh sách các phần tử.

- ✓ Class **CommandBatchUpdateCallback**

Ta phải implement cho interface **ICommandCallback** bằng cách tạo thêm 1 class **CommandBatchUpdateCallback** như sau:

```
1 reference
public class CommandBatchUpdateCallback : ICommandCallback<int>
{
    IBatchUpdateCommandSetter setter;
    IDynamicParameter parameter;
    1 reference
    public CommandBatchUpdateCallback(IBatchUpdateCommandSetter setter, IDynamicParameter parameter)
    {
        this.setter = setter;
        this.parameter = parameter;
    }
    6 references
    public int Handle(IDbCommand command)
    {
        try
        {
            parameter.Handle(command);
            int c = 0;
            int batchSize = setter.BatchSize;
            for (int i = 0; i < batchSize; i++)
            {
                setter.SetValues(parameter, i);
                c += command.ExecuteNonQuery();
            }
            return c;
        }catch(Exception ex)
        {
            throw new Exception(ex.Message);
        }
    }
}
```

Figure 52: class CommandBatchUpdateCallback

Trong phương thức **Handle** ta cần lưu ý rằng tham số **Parameter** cần phải thực hiện trước để thiết lập từng tham số cho câu truy vấn và sau đó trong vòng lặp for ta sẽ thực hiện thiết lập giá trị cho tham số tương ứng rồi mới gọi hàm **ExecuteNoneQuery** để ghi nhận tham số.

- ✓ Method **BatchUpdate**

Tiếp theo sau ta vào interface **IDbContext** cập nhật thêm phương thức **BatchUpdate** như sau

```
6 references
public interface IDbContext
{
    2 references
    int BatchUpdate(string sql, IDynamicParameter parameter, IBatchUpdateCommandSetter setter, CommandType commandType = CommandType.Text);
    2 references
    IQueryMultiple QueryMultiple(string sql, IDynamicParameter parameter, CommandType commandType = CommandType.Text);
    2 references
    IQueryMultiple QueryMultiple(string sql, ICommandSetter setter = null, CommandType commandType = CommandType.Text);
    1 reference
    T Scalar<T>(string sql, IDynamicParameter parameter, CommandType commandType = CommandType.Text);
    3 references
    T Scalar<T>(string sql, ICommandSetter setter = null, CommandType commandType = CommandType.Text);
    2 references
    int Update(string sql, IDynamicParameter parameter, CommandType commandType = CommandType.Text);
    2 references
    int Update(string sql, ICommandSetter setter, CommandType commandType = CommandType.Text);
    2 references
    T QueryOne<T>(string sql, IDynamicParameter parameter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text);
    1 reference
    List<T> Query<T>(string sql, IDynamicParameter parameter, IRowMapper<T> mapper, CommandType commandType = CommandType.Text);
```

Figure 53: method BatchUpdate

Và cũng phải thêm vào class **DbContext** phương thức **BatchUpdate** như hình sau:

```
1 reference
public class DbContext : IDbContext
{
    IDbConnection connection;
    1 reference
    public DbContext(IDbConnection connection)...
    2 references
    public int BatchUpdate(string sql, IDynamicParameter parameter, IBatchUpdateCommandSetter setter, CommandType commandType = CommandType.Text)
    {
        return Execute(sql, new CommandBatchUpdateCallback(setter, parameter), commandType);
    }
    2 references
    public IQueryMultiple QueryMultiple(string sql, IDynamicParameter parameter, CommandType commandType = CommandType.Text)...
    2 references
    public IQueryMultiple QueryMultiple(string sql, ICommandSetter setter = null, CommandType commandType = CommandType.Text)...
```

Figure 54: implement method BatchUpdate

Trong phương thức này ta cũng khởi tạo class **CommandBatchUpdateCallback** và gọi hàm **Execute** để hoàn thành hàm **ExecuteUpdate**.

Ta xét thử 1 ví dụ là delete nhiều dòng dữ liệu trên bảng **Role** cách thực hiện như sau

- ✓ Class **DeleteBatch**

Đầu tiên ta phải tạo 1 class **DeleteBatch** implement cho interface **IBatchUpdateCommandSetter** vừa tạo ở trên

```
1 reference
class DeleteBatch : IBatchUpdateCommandSetter
{
    Guid[] arr;
    1 reference
    public DeleteBatch(Guid[] arr)
    {
        this.arr = arr;
    }
    2 references
    public int BatchSize { get { return arr.Length; } }

    2 references
    public void SetValues(IDynamicParameter parameter, int i)
    {
        parameter.SetValue("@Id", arr[i]);
    }
}
```

Figure 55: class DeleteBatch

Trong class này ta phải truyền vào một array RoleId và chỉ cần chỉ ra số lượng phần tử của array và trong phương thức ta chỉ cần thiết lập giá trị cho tham số @Id.

- ✓ Method **Delete**

Tiếp sau ta thêm phương thức **Delete** vào trong lớp **RoleRepository** như sau:

```
public class RoleRepository : BaseRepository
{
    1 reference
    public RoleRepository(IDbContext dbContext)...
    1 reference
    public List<Role> GetRoles()...
    0 references
    public Role GetRole(int id)...
    1 reference
    public int Add(Role obj)...
    1 reference
    public int Count()...
    1 reference
    public int Delete(Guid[] arr)
    {
        IDynamicParameter parameter = new DynamicParameter(1);
        parameter.Add("@Id", dbType: DbType.Guid);
        return dbContext.BatchUpdate("DELETE FROM Role WHERE RoleId = @Id", parameter, new DeleteBatch(arr));
    }
    /* readonly static string connectionString = "Data Source=.;Initial Catalog=BikeStore;Integrated Security=True"; ...
}
```

Figure 56: Method Delete

Trong phương thức delete này ta phải khởi tạo class **DeleteBatch** và truyền vào array id. Sau đó gọi hàm **BatchUpdate** là mọi thứ sẽ để hàm này giải quyết một cách nhanh gọn.