

# Blazor Server

## Dapper

2023

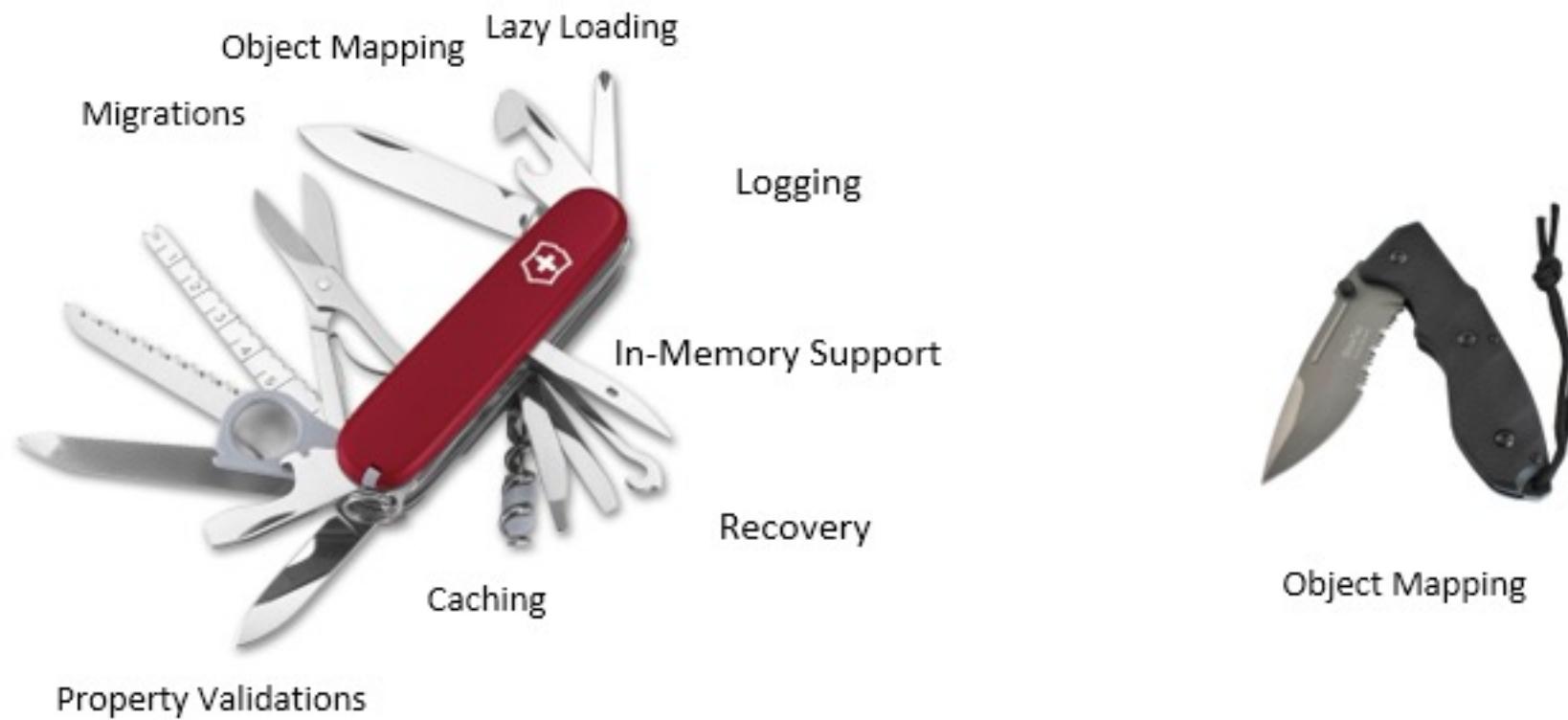
# Introduction

- Dapper is a **Micro-ORM** which helps to map plain **query** output to **domain** classes.
- It can run plain SQL queries with **great performance**.
- Dapper is a **lightweight** framework that supports all major databases like SQLite, Firebird, Oracle, MySQL, and SQL Server.
- It does not have database-specific **implementation**.
- Dapper is built by the StackOverflow team and released as **open source**.

# Create a Class Library

- Code reusability is very important in the software development process.
- Using the already-written code can save time and resources and reduce redundancy.
- A Class library is a good example of code reusability.
- In object-oriented programming, a class library is a collection of prewritten classes or coded templates which contains the related code.
- When we finish a class library, we can decide whether you want to distribute it as a third-party component or whether you want to include it as a DLL with one or more applications.

# Traditional ORM [Entity Framework] vs Micro ORM



# Traditional ORM [Entity Framework] vs Micro ORM

- Performance of SELECT mapping over 500 iterations

Method	Duration
Hand coded (using a SqlDataReader)	47ms
Dapper	49ms
PetaPoco	52ms
NHibernate SQL	104ms
Entity framework (ExecuteStoreQuery)	631ms

# Database setup

- Create a **Cards Database** and run the following script:

```
DROP TABLE IF EXISTS
CREATE TABLE users
    int PRIMARY KEY IDENTITY
name varchar 200
    varchar 300 UNIQUE

DROP TABLE IF EXISTS
CREATE TABLE cards
    int PRIMARY KEY IDENTITY
    varchar 300
name varchar 200 NOT NULL
description varchar 300
image varchar
date datetime NOT NULL
    int NOT NULL
owner int NOT NULL
    int
FOREIGN KEY owner REFERENCES
```

```
DROP TABLE IF EXISTS
CREATE TABLE tags
    int PRIMARY KEY IDENTITY
name varchar 100 NOT NULL

DROP TABLE IF EXISTS
CREATE TABLE card_tags
    int
    int
PRIMARY KEY
FOREIGN KEY
FOREIGN KEY
    REFERENCES
    REFERENCES
```

# Create a Class Library

- [Create](#) the class library in the solution:
  - `dotnet new classlib -o DataLayer`
- [Add Reference](#) to the main Project:
  - `dotnet add app/app.csproj reference DataLayer/DataLayer.csproj`

## Dapper setup

- Install [Dapper](#) in the [class library](#):
  - `dotnet add package Dapper --version 2.0.123`
- Additionally, install the following package:
  - `dotnet add package Microsoft.Extensions.Configuration --version 7.0.0`

# AppSettings

- Add to the `appsettings.json` file the connection string to access your SQL Server database:

```
"ConnectionStrings": {  
    "Default": "(...)"  
}
```

# SqlDataAccess

- Let's create a [class](#) to encapsulate the logic behind configuring access to database:
- The purpose of this class, [SqlDataAccess](#), is to provide a means of accessing a SQL database and performing data operations.
- It implements the [ISqlDataAccess](#) [interface](#), indicating that it adheres to a specific contract defined by that interface.

# SqlDataAccess

```
public class SqlDataAccess : ISqlDataAccess{
    private readonly IConfiguration _config;
    public string ConnectionStringName { get; set; } = "Default";
    public SqlDataAccess(IConfiguration config){
        _config = config;
    }
    public async Task<List<T>> LoadData<T, U>(string sql, U parameters){
        string? connectionString = _config.GetConnectionString(ConnectionStringName);
        using ( IDbConnection connection = new SqlConnection(connectionString)){
            var data = await connection.QueryAsync<T>(sql, parameters);
            return data.ToList();
        }
    }
    public async Task SaveData<T>(string sql, T parameters){
        string? connectionString = _config.GetConnectionString(ConnectionStringName);
        using ( IDbConnection connection = new SqlConnection(connectionString)){
            await connection.ExecuteAsync(sql, parameters);
        }
    }
}
```

LoadData<T, U> is used to execute a SQL query and return the resulting data as a list of objects of type T.

SaveData method executes an SQL query asynchronously to save data to the database. It retrieves the connection string from the configuration

# SqlDataAccess

- Let's create an interface for the `SqlDataAccess` class:

```
namespace DataLayer;

public interface ISqlDataAccess{
    string ConnectionStringName { get; set; }
    Task<List<T>> LoadData<T, U>(string sql, U parameters);
    Task SaveData<T>(string sql, T parameters);
}
```

# SqlDataAccess

- Let's now create **CardModel** DTO to support data handling:

```
namespace DataLayer;
public record CardModel
{
    public string? _Id { get; set; }
    public int Id { get; set; }
    public string Name { get; set; } = "";
    public string Description { get; set; } = "";
    public string Image { get; set; } = "";
    public DateTime? Date { get; set; }
    public int TimeInMinutes { get; set; }
    public UserModel? Owner { get; set; } = new UserModel();

    public List<TagModel>? Tags { get; set; } = new List<TagModel>();
}
```

The CardModel.cs class is a Data Transfer Object (DTO) since it is primarily used for transferring data between the class library and the Blazor app.

# SqlDataAccess

- Let's now create **TagModel** DTO to support data handling:

```
namespace DataLayer;

public record TagModel
{
    public int Id { get; set; }

    public string Name { get; set; } = "";
}
```

# SqlDataAccess

- Let's now create `User` DTO to support data handling:

```
namespace DataLayer;
public record UserModel
{
    int Id { get; }
    string _Id { get; set; } = "";
    string Name { get; set; } = "";
    string Email { get; set; } = "";
}
```

# Repository Skeleton

- We are using a **repository pattern** to **encapsulate** data access.
- Many people like the repository pattern because it provides good separation of concerns, **easy mocking**, and **good encapsulation**.
- It's going to allow to easily **swap** out different implementations
- The idea with this pattern is to have a **generic abstract** way for the app to work with the **data layer** without being bothered if the implementation is towards a local database or towards an online API.

# Repository Skeleton

- Let's create the interface for the [repository](#):

```
namespace DataLayer;

public interface ICardRepository{
    Task<CardModel> Find(int id);

    Task<List<CardModel>> FindAll();

    Task Update(CardModel card);

    Task Remove(int id);

    Task<CardModel> Add(CardModel card);
}
```

# Repository Skeleton

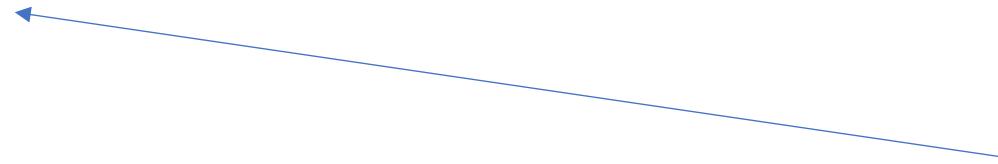
- Let's implement the class with, for now, the (limited) `findAll` behavior:

```
namespace DataLayer;
public class CardRepository : ICardRepository{
    private ISqlDataAccess _db;
    public CardRepository(ISqlDataAccess db){
        _db = db;
    }
    public Task<CardModel> Find(int id){
        throw new NotImplementedException();
    }
    public Task<List<CardModel>> GetAll(){
        string sql = "select * from Cards";
        return _db.LoadData<CardModel, dynamic>(sql, new { });
    }
    ...
}
```

## Configure main app – Program.cs

- Let's add to [Program.cs](#) file the SQLDataAcess and the repository:

```
builder.Services.AddTransient<ISqlDataAccess, SqlDataAccess>();  
builder.Services.AddTransient<ICardRepository, CardRepository>();
```



Transient objects are always different; a new instance is provided to every controller and every service

# Configure main app – Program.cs

- Now we can [test](#) if it works!
- Modify FetchData.razor do see the results.
- Example:

Cards List						
This component demonstrates fetching data from a database.						
id	name	description	Owner	Tags	Test Update	Test Delete
15	Test Card	Test Description	Bruno	Tag 1 Tag 2 Tag 3	Test Update card with tags	Delete Test Data
17	Test Card	Test Description	Bruno	Tag 1 Tag 2 Tag 3	Test Update card with tags	Delete Test Data
18	Test Card	Test Description	Bruno	Tag 1 Tag 2 Tag 3	Test Update card with tags	Delete Test Data

[Test Add card with tags](#)

## Configure main app – Program.cs

- Now we can **test** if it works!
- In **FetchData.razor** page changes fetch data from the database:

```
@page "/fetchdata"
@using DataLayer
@inject ICardRepository _db
(...)

@if (cards == null){
    (...) <p><em>Loading...</em></p>
} else{
    <table class="table"><thead>

@code {
    private List<DataLayer.CardModel> cards;

    protected override async Task OnInitializedAsync()
    {
        cards = await _db.FindAll();
    }
}
```

# Loading data with filter

- Let's create the `find` method (`CardRepository`) for a `Card`:

```
public async Task<CardModel> Find(int id)
{
    string sql = "SELECT * FROM Cards WHERE id = @Id";
    var card = await _db.LoadData<CardModel, dynamic>(sql, new { Id = id });
    return card.SingleOrDefault();
}
```

# Adding data

- Let's create the `Add` method (`CardRepository`):

```
public async Task<CardModel> Add(CardModel card)
{
    string sql =
        @"INSERT INTO dbo.cards(name,description,image, date, time_in_minutes, owner)
        VALUES(@Name,@Description, @Image, @Date, @TimeInMinutes,1);"
        + "SELECT CAST(SCOPE_IDENTITY() as int)";
    int id = await _db.SaveDataWithReturn<int, CardModel>(sql, card);
    return new CardModel()
    {
        Id = id,
        Name = card.Name,
        Description = card.Description,
        Image = card.Image,
        Date = card.Date,
        TimeInMinutes = card.TimeInMinutes
    };
}
```

For now, we are hardcoding ownerId

Records are immutable

We are executing two statements, the first one to insert and the second one to retrieve the inserted id

# Update Data

- Let's create the `update` method (`CardRepository`):

```
public async Task Update(CardModel card)
{
    string sql =
        "UPDATE dbo.cards SET name = @Name, description = @Description, image = @Image, date =
        @Date, time_in_minutes = @TimeInMinutes WHERE id = @Id";
    await _db.SaveData<CardModel>(sql, card);
}
```

- Modify `FectData.razor` and test `FindAll`, `Find`, `Add`, `Update` for simple objects.
- Implement the `Remove`.

# Using Relationships With Dapper

- Notice that we don't have the `Owner` data or the `Tags` data.
- Dapper provides a feature called `Multi mapping` to map data to `multiple` objects explicitly and nested objects.
- The `splitOn` parameter tells Dapper what column(s) to use to `split` the data into `multiple` objects.

# Using Relationships With Dapper - One to Many

- Let's add `LoadData` to `SqlDataAccess` and `ISqlDataAccess` for loading `Cards` and the `User`.

```
Task<List<T1>> LoadData<T1, T2, U>(  
    string sql,  
    U parameters,  
    Func<T1, T2, T1> mappingFunction,  
    string splitOn,  
    Func<T1, object> groupByFunc = null,  
    Func<IGrouping<object, T1>, T1> selectFunc = null  
) ;
```

`Func<T1, T2, T1> mappingFunction`: This parameter is a delegate (function) that represents a mapping function that is responsible for mapping the data retrieved from the data source into an instance of type `T1`.

`Func<T1, object> groupByFunc = null`: It represents a grouping function that can be used to group the data by a specific property. It is set to `null` by default, indicating that grouping is not required.

`<T1, T2, U>`: These are generic type parameters. They allow the method to work with different types without specifying them directly. `T1` and `T2` represent the types of objects that the method will operate on, while `U` represents the type of the `parameters` parameter.

`Func<IGrouping<object, T1>, T1> selectFunc = null`: This parameter is another optional delegate (function) that takes an `IGrouping<object, T1>` and returns an object of type `T1`. It represents a selection function that can be used to perform a final selection on the grouped data.

# Using Relationships With Dapper – One to Many

- Let's create `FindAll` method to return `Cards` and `Users`

```
public async Task<List<CardModel>> FindAll()
{
    string sql = "SELECT a.id, a.name, description, image, date, time_in_minutes, b.id, b.name,
    b.email
        FROM [dbo].[cards] AS a JOIN dbo.users AS b ON a.[owner]=b.id ";
    return await _db.LoadData<CardModel, UserModel, dynamic>(
        sql,
        new { },
        (card, user) =>{
            card.Owner = user;
            return card;
        },
        "Id"
    );
}
```

- a. The `sql` variable contains the SQL query to execute.
- b. The second argument is an anonymous object `{ }` representing any parameters that might be passed to the SQL query.
- c. The third argument is a lambda function `(card, user) => { ... }` that maps the result rows to `CardModel` and `UserModel` objects. Inside the lambda function, the `card` and `user` parameters represent the columns retrieved from the respective tables.
- d. The fourth argument is a string "`Id`", which specifies the property to be used as the identifier for the parent-child relationship. It indicates that the `Id` property of the `CardModel` should be used.

# Using Relationships With Dapper Many to Many

- The multi-mapping works with **multiple** relationships. we are loading **Cards**, **Tags** and **User**

```
public async Task<List<CardModel>> FindAll(){
    string sql = @"SELECT a.id, a.name, description, image, date, time_in_minutes, u.id,
u.name, u.email, c.id, c.name
                FROM [dbo].[cards] AS a LEFT JOIN dbo.card_tags AS b ON a.[id]=b.id_card
                                LEFT JOIN dbo.tags AS c ON b.id_tag = c.id
                                LEFT JOIN dbo.users AS u ON u.id = a.[owner]";
    return await _db.LoadData<CardModel, UserModel, TagModel, dynamic>(sql,new { },(card,
owner, tag) =>{
        card.Owner= owner;
        card.Tags.Add(tag);
        return card;
    },
    "id, id",
    p => p.Id,
    g =>{
        var groupedCard = g.First();
        groupedCard.Tags = g.Select(p => p.Tags.Single()).ToList();
        return groupedCard;
    });
}
```

specifying the column names to be used for grouping the results.

specifies how to extract the ID from a grouped set of records.

responsible for grouping the cards and tags based on the specified column names

# Using Relationships With Dapper Many to Many

- Let's add the new method to `SqlDataAccess`:

```
public async Task<List<T1>> LoadData<T1, T2, T3, U>(
    string sql,
    U parameters,
    Func<T1, T2, T3, T1> mappingFunction,
    string splitOn,
    Func<T1, object> groupByFunc = null,
    Func<IGrouping<object, T1>, T1> selectFunc = null
){
    string? connectionString = _config.GetConnectionString(ConnectionStringName);
    using ( IDbConnection connection = new SqlConnection(connectionString)){
        var data = await connection.QueryAsync<T1, T2, T3,
        T1>(sql,mappingFunction,splitOn: splitOn);
        if (groupByFunc != null && selectFunc != null){
            var result = data.GroupBy(groupByFunc).Select(selectFunc);
            return result.ToList();
        }
        return data.ToList();
    }
}
```

# Using Relationships With Dapper Many to Many

- Let's add the new method to `ISqlDataAccess`:

```
Task<List<T1>> LoadData<T1, T2, T3, U>(
    string sql,
    U parameters,
    Func<T1, T2, T3, T1> mappingFunction,
    string splitOn,
    Func<T1, object> groupByFunc = null,
    Func<IGrouping<object, T1>, T1> selectFunc = null
);
```

- Test these new methods using `FetchData.razor`

# Updating Find (one) using Multiple results

- The Dapper [QueryMultiple](#) method allows you to select [multiple results](#) from a database query.
- That feature is very useful for selecting [multiple result sets](#) at once, thus avoiding unnecessary round trips to the database server.
- Let's add a new method to [SqlDataAccess](#) and respective interface in [ISqlDataAccess](#) :

```
public async Task<Tuple<T1, List<T2>, T3>> LoadData<T1 , T2, T3, U>(string sql, U parameters){  
    string? connectionString = _config.GetConnectionString(ConnectionStringName);  
  
    using ( IDbConnection connection = new SqlConnection(connectionString)){  
        using(var multipleResults = await connection.QueryMultipleAsync(sql, parameters)){  
            var _t1 = multipleResults.Read<T1>();  
            var _t2 = multipleResults.Read<T2>().ToList();  
            var _t3 = multipleResults.Read<T3>();  
            if(_t1 != null && _t2 != null){  
                return new Tuple<T1, List<T2>, T3>(_t1.SingleOrDefault(), _t2,_t3.SingleOrDefault());  
            }  
        }  
    }  
    return null;  
}
```

# Updating Find (one) using Multiple results

- Let's change Find method in CardRepository to get Card, User and Tags using QueryMultiple method

```
public async Task<CardModel> Find(int id){  
    string sql =  
        "SELECT * FROM Users WHERE id = (SELECT owner FROM Cards WHERE id=@Id);"  
        + "SELECT * FROM Card_Tags WHERE id_card = @Id;"  
        + "SELECT id,name,description,image,date, time_in_minutes FROM Cards WHERE id = @Id;";  
  
    Tuple<UserModel, List<TagModel>, CardModel> result = await this._db.LoadData<  
        UserModel,  
        TagModel,  
        CardModel,  
        dynamic  
>(sql, new { Id = id });  
  
    CardModel card = result.Item3;  
    card.Tags.AddRange(result.Item2);  
    card.Owner = result.Item1;  
    return card;  
}
```

Queries should match the types

# Create-update-delete with complex objects

- **Add** method will deal with **simple objects** while **save** method will deal with **complex objects**
- First we are going to create a **new field** in **CardModel** and **Tag** class (we are assuming the card owner always exist and was not changed in this context):

```
public bool IsNew => this.Id == default(int);
```

Default value for int is 0  
If it is 0, the record is new

## Create-update-delete with complex objects

- In `TagModel` we are adding `IsNew` and `IsDeleted` properties to control when `Tags` are added or removed from a card

```
public bool IsNew {get;set;} = true;  
public bool IsDeleted { get; set; } = false;
```

# Create-update-delete with complex objects

- We are creating a new method in **CardRepository** named: **Save**;
- Before that, let's create **TagRepository** class with methods for **Add Tag**, **Update**, **Delete** and **Find Tag** :

```
namespace DataLayer;  
  
public interface ITagRepository  
{  
    Task<TagModel> Add(TagModel tag);  
  
    Task Update(TagModel tag, CardModel car);  
  
    Task Delete(TagModel tag, CardModel car);  
  
    Task<TagModel> Find(TagModel tag);  
}
```

# Create-update-delete with complex objects

- TagRepository

```
namespace DataLayer;  
  
public class TagRepository : ITagRepository  
{  
    private ISqlDataAccess _db;  
  
    public TagRepository(ISqlDataAccess db)  
    {  
        _db = db;  
    }  
    (...)  
}
```

# Create-update-delete with complex objects

- Add method

```
public async Task<TagModel> Add(TagModel tag)
{
    string sql =
        @"INSERT INTO dbo.tags(name) VALUES(@name);" +
        "SELECT CAST(SCOPE_IDENTITY() as int)";
    int id = await _db.SaveData<int, TagModel>(sql, tag);
    return new TagModel() { Id = id, Name = tag.Name };
}
```

- Update

```
public async Task Update(TagModel tag, CardModel car)
{
    string sql = "INSERT INTO dbo.card_tags(id_card, id_tag) VALUES(@id_card, @id_tag)";
    await _db.SaveData<dynamic>(sql, new { id_card = car.Id, id_tag = tag.Id });
}
```

# Create-update-delete with complex objects

- Delete method

```
public async Task Delete(TagModel tag, CardModel car)
{
    string sql = "DELETE FROM dbo.card_tags WHERE id_card = @id_card AND id_tag = @id_tag";
    await _db.SaveData<dynamic>(sql, new { id_card = car.Id, id_tag = tag.Id });
}
```

Ensure you have ON  
DELETE CASCADE in the  
FK

- Find method

```
public async Task<TagModel> Find(TagModel tag)
{
    string sql = "SELECT * FROM Tags WHERE id = @Id";
    var returnedTag = await _db.LoadData<TagModel, dynamic>(sql, new { Id = tag.Id });
    return returnedTag.FirstOrDefault();
}
```

## Create-update-delete with complex objects

- Let's change `CardRepository` to use `TagRepository` instance:

```
(...)
private ITagRepository _tagRepository;

public CardRepository(ISqlDataAccess db)
{
    _db = db;
    _tagRepository = new TagRepository(db);
}
(...)
```

# Create-update-delete with complex objects

- Let's create **Save** method in the **CardRepository**

```
public async Task<CardModel> Save(CardModel card){  
    if (card.IsNew){  
        CardModel result = await this.Add(card);  
        card.Id = result.Id;  
    }else{  
        await this.Update(card);  
    }  
    List<TagModel> updatedTags = new List<TagModel>();  
    foreach (var tag in card.Tags){  
        if (tag.IsNew){  
            TagModel result = await _tagRepository.Add(tag);  
            await _tagRepository.Update(result, card);  
            updatedTags.Add(result);  
        }else if (!tag.IsDeleted){  
            updatedTags.Add(tag);  
        }  
    }  
    (...)
```

# Create-update-delete with complex objects

- Let's create **Save** method in the **CardRepository**

```
(...)  
foreach (var tag in card.Tags.Where(t => t.IsDeleted)){  
    await _tagRepository.Delete(tag, card);  
}  
card.Tags = updatedTags;  
return card;  
}
```

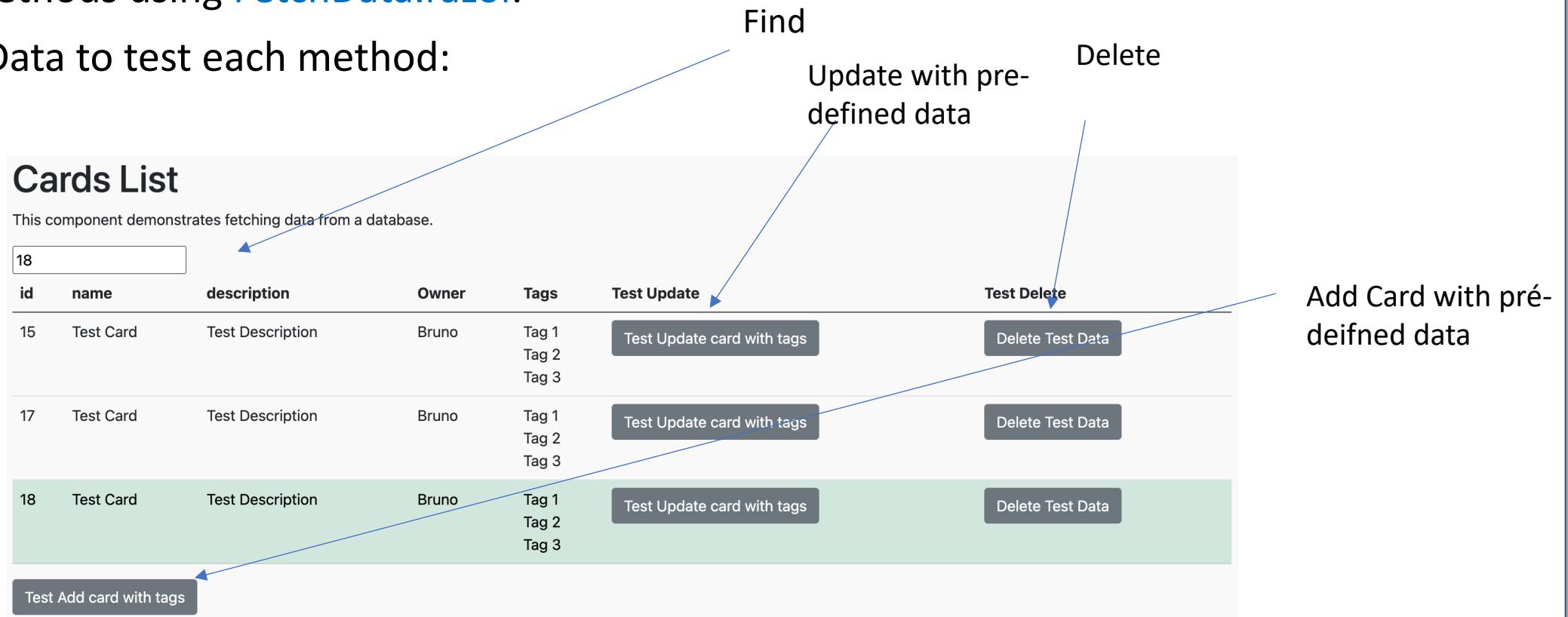
## Create-update-delete with complex objects

- Everything may have worked just now, but remember, we're updating **multiple rows** here.
- So we need to wrap this method in a **transaction** in case anything goes wrong.
- Let's use a TransactionScope for this.

```
public async Task<CardModel> Save(CardModel card)
{
    using (var txScope = new
        TransactionScope(TransactionScopeAsyncFlowOption.Enabled)){
        (...)
```

# Create-update-delete with complex objects

- Test all methods using [FetchData.razor](#).
- Use test Data to test each method:



## Summary

- Dapper is a **Micro-ORM** which helps to map **plain query** output to **domain** classes.
- It can run plain SQL queries with **great performance**.
- A **class library** is used for reusability and separation of concerns;
- A **DataAcess library** can be used to separate application logic and data layer

## Summary

- The repository pattern provides good separation of concerns, [easy mocking](#), and [good encapsulation](#).
- Dapper provides several ways for [mapping objects](#), [QueryMultiple](#) is one of those methods

# Blazor Server

## Dapper

2023