Blazor Server

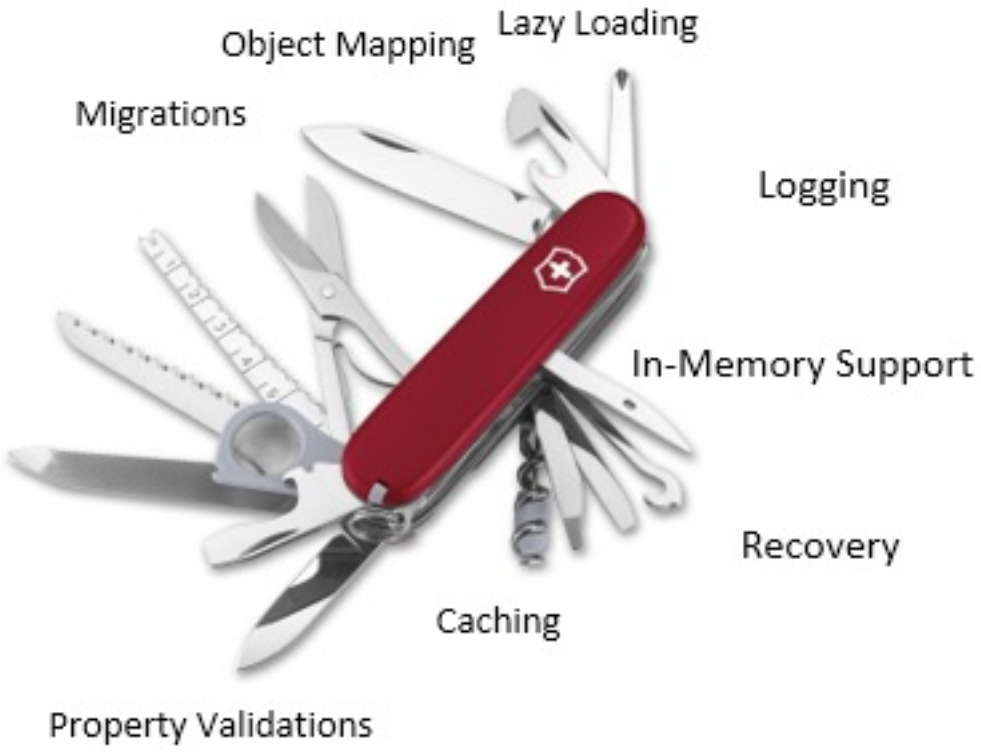
Dapper

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.

Traditional ORM [Entity Framework] vs Micro ORM



Object Mapping

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           REFERENCES
FOREIGN KEY          REFERENCES
```

https://gist.github.com/brunobmo/df1c0096080dbb4eac2c8bbc9afed199

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.

http://www.authorcode.com/creating-a-class-library-with-c-and-net-core-in-visual-studio-code/

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": ”(...)"
}
```

SqlDataAcess

- Let’s create a class to encapsulate the logic behind configuring acess 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.

SqlDataAcess

```
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

https://gist.github.com/brunobmo/6400ce751f23c1947bd1bb9c0d610992

SqlDataAcess

- Let’s create an interface for the SqlDataAcess 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);
}
```

https://gist.github.com/brunobmo/6d4d56757b5ea1b44b6f3793b469da47

SqlDataAcess

- 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.

https://gist.github.com/brunobmo/1047be85df7935f796f9198c9eaf1495

SqlDataAcess

- 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; } = "";
}
```

https://gist.github.com/brunobmo/f1a3baec1cf929c51f0535ddfa48aa4c

SqlDataAcess

- 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;} = "";
}
```

https://gist.github.com/brunobmo/ce3e780fce276c836bd9ba30d3a77070

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.

https://medium.com/@pererikbergman/repository-design-pattern-e28c0f3e4a30

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);
}
```

https://gist.github.com/brunobmo/5fd3a4728161e20c73ffa738f0d65496

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>> FindAll(){
     string sql = "select * from Cards";
     return _db.LoadData<CardModel, dynamic>(sql, new { });
   }
(...)
```

https://gist.github.com/brunobmo/d18a372636ac170caf8bbb71897a8c44

Configure main app – Program.cs

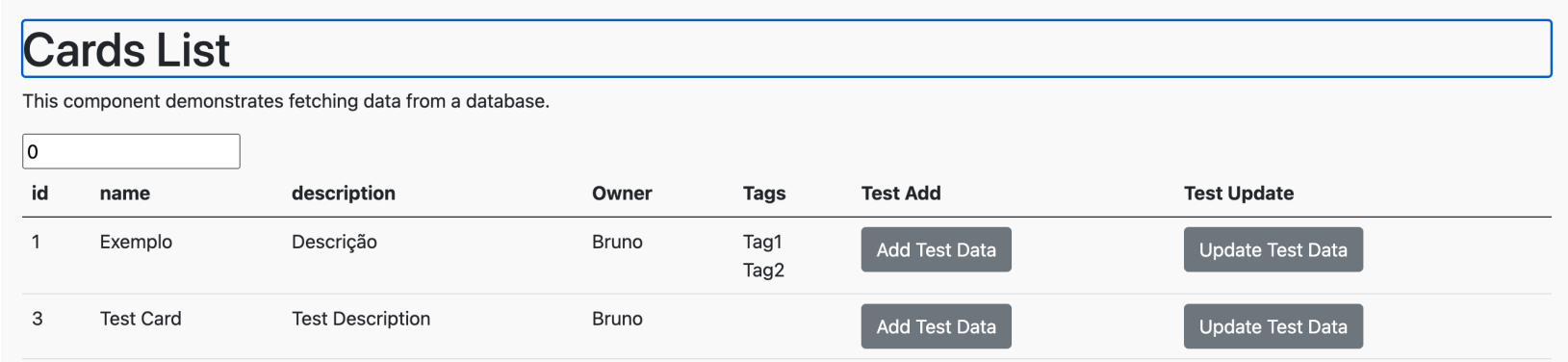
- Let's add to Program.cs file the SQLDataAcess and the repository:

```
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:



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();
     }
}
```

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

```
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.LoadDataParentChild<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.

https://gist.github.com/brunobmo/442633482221029c6f6a4dec2d5bf866

Using Relationships With Dapper - One to Many

- Let's add LoadDataParentChild to ISqlDataAccess:

```
Task<List<T1>> LoadDataParentChild<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
);
```

<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<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.

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

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.

https://gist.github.com/brunobmo/9a5ec86ea0b6488c5ea2f32b0da69300

Using Relationships With Dapper Many to Many

- Multiple mapping also works with many-to-many relationships.

```
public async Task<List<CardModel>> FindAll(){
   string sql ="SELECT a.id, a.name, description, image, date, time_in_minutes, c.id AS tagId,
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";
return await _db.LoadDataParentChild<CardModel, UserModel , TagModel, dynamic>(sql,new { },
    (card, tag) =>{
      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

https://gist.github.com/brunobmo/c5d836df677711aedf74461ef9607b15

Using Relationships With Dapper Many to Many

- The multi-mapping works with multiple relationships.

```
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.LoadDataParentChild<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;
    });
}
```

https://gist.github.com/brunobmo/b3d589410f220d3bdc13fe5821c35500

Using Relationships With Dapper Many to Many

- Let's add the new method to SqlDataAccess:

```
Task<List<T1>> LoadDataParentChild<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
);
```

https://gist.github.com/brunobmo/52f3f7a4642c60942d06deb74b8c66cd

Using Relationships With Dapper Many to Many

- 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);
}
```

https://gist.github.com/brunobmo/4b33ecdfc9a13047de7390557d877ade

Using Relationships With Dapper Many to Many

- 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
    };
}
```

https://gist.github.com/brunobmo/7021ac189b3fc2f66be4261c3431e59f

Blazor Server

Dapper

2023