Blazor

Database and Dapper

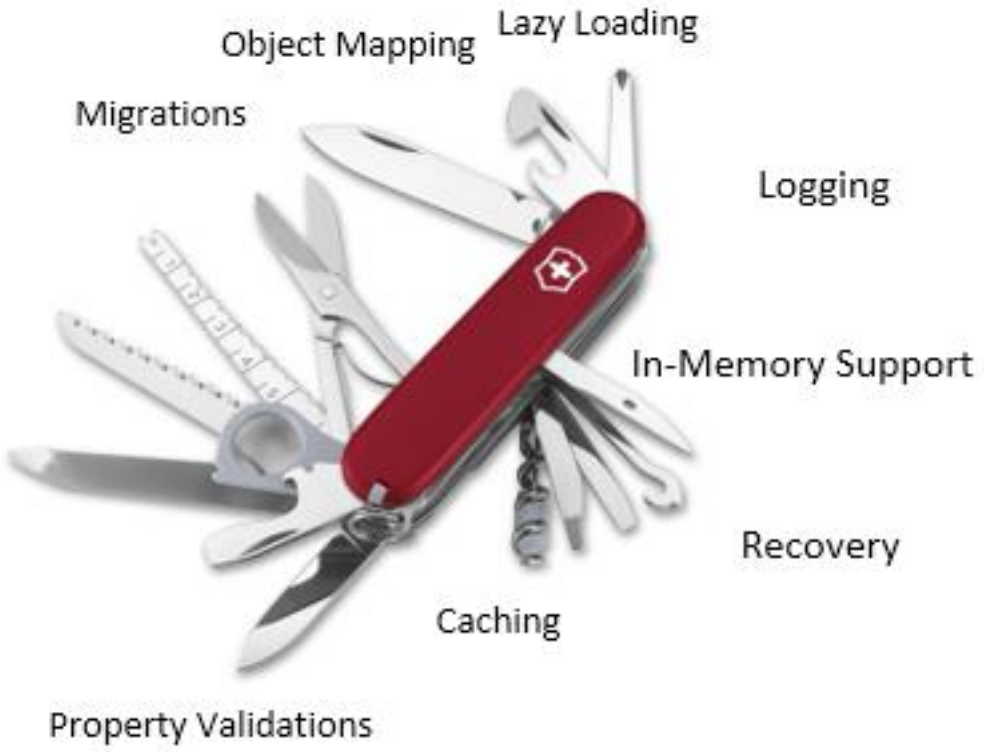
Bruno Oliveira - bruno.oliveira3@sapo.pt

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.
- All you need is a valid and open connection.
- Dapper is built by 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 users;
CREATE TABLE users (
  id int PRIMARY KEY IDENTITY,
  name varchar(200),
  email varchar(300) UNIQUE
);
DROP TABLE IF EXISTS cards;
CREATE TABLE cards (
   id int PRIMARY KEY IDENTITY,
  _id varchar(300),
  name varchar(200) NOT NULL,
  description varchar(300),
  image varchar(max),
  date datetime NOT NULL,
  time_in_minutes int NOT NULL,
  owner int NOT NULL,
  image_action int,
FOREIGN KEY (owner) REFERENCES users(id)
);
```

```
DROP TABLE IF EXISTS tags;
CREATE TABLE tags(
  id int PRIMARY KEY IDENTITY,
  name varchar(100) NOT NULL
)
DROP TABLE IF EXISTS card_tags;
CREATE TABLE card_tags(
  id_card int,
  id_tag int,
  PRIMARY KEY(id_card,id_tag),
  FOREIGN KEY (id_card) REFERENCES cards(id),
  FOREIGN KEY (id_tag) REFERENCES tags(id)
)
```

https://gist.github.com/brunobmo/70340bb3ca0558f4eaa52a29b36a120b

Create a Class Library

- Code reusability is very important in the software development process.
- 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

- Outside project folder
- dotnet new classlib -o DataLayer
- Add Reference to 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
  - dotnet add package System.Data.SqlClient --version 4.8.5
- 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 access to database:

```
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);
     }
  }
}
```

https://gist.github.com/brunobmo/884b8f238321b7358bb276fd02350429

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/a4c1ab077426036231b9d7ca07145fa2

SqlDataAcess

- Let’s now create DTOs to support data handling:

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

https://gist.github.com/brunobmo/8a2890a7517974d1c316c3b24fb6c26a

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 an online API.being bothered

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<CardModel> Update(CardModel card);
  Task Remove(int id);
}
```

https://gist.github.com/brunobmo/2e66778da07f126c8cc4ce47c0bfe3bc

Repository Skeleton

- Let's implement the class with, for now, the 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/90cc7a6426cb7f33536d898b2951f2d9

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!
- 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();
        }
}
```

https://gist.github.com/brunobmo/7eafb8f2f719b36ff4d506d54ab0e6d1

More guides

- Available at: https://github.com/brunobmo/BlazorCourseNet7/tree/main

https://gist.github.com/brunobmo/7eafb8f2f719b36ff4d506d54ab0e6d1

Blazor

Database and Dapper

Bruno Oliveira - bruno.oliveira3@sapo.pt

2023