Practice Day 1

Concepts

- "F1 Emu" is a basic solution to manage Formula 1 Lap/Sector Times into a database
- Each lap info record could represent a partial time of the lap (a sector) or an entire lap time
- To be easy, each lap has always 3 sectors. This means that when a record has the number 3 in the sector, it also represents the entire lap time.

Hands On

1. Clone the **MinimalApis** repository **https://github.com/hjmo-lab/minimal-apis.git**.
2. Run the project F1Emu.Api to ensure that all the requirements are set.
3. Add a "Lap time" to the database

   3.1. Create the request

   Inside the Domain/Laps folder, create a new class **AddLapRequest** that should be the request object to insert a new Lap info into the database according to the following required fields:

Property	**Type**	**Description**
DriverName	string	The name driver
LapIndex	int	Number of the lap
Sector	int	Current Sector number
ElsapsedTime	string	The elapsed time including the current sector in the mm:ss.SSS string format
SectorTime	string	The elapsed time only for the current sector in the mm:ss.SSS string format

   3.2. Create Post method with route "/laps": create and fill domain entity

```
app.MapPost("/laps", async (AddLapRequest addLapRequest) =>
{
    Lap lap = new()
    {
        Id = Guid.NewGuid(),
        DriverName = addLapRequest.DriverName,
        LapIndex = addLapRequest.LapIndex,
        Sector = addLapRequest.Sector,
        CreatedOn = DateTime.UtcNow,
        ElapsedTime = TimeSpan.Parse($"00:{addLapRequest.ElapsedTime}"),
        SectorTime = TimeSpan.Parse($"00:{addLapRequest.SectorTime}")
    };
    return Results.Ok();
});
```

3.3. Define repository Add method to insert data.

3.3.1. Let’s start to define the same entity, but in the data side. Go to the **LapDataModel** class (Infrastructure/SqlLap/LapDataModel.cs) and define the following properties:

```
public class LapDataModel
 {
     public Guid Id { get; set; }
     public string DriverName { get; set; }
     public int LapIndex { get; set; }
     public int Sector { get; set; }
     public int Minutes { get; set; }
     public int Seconds { get; set; }
     public int Milliseconds { get; set; }
     public long ElapsedTime { get; set; }
     public long SectorTime { get; set; }
     public DateTime CreatedOn { get; set; }
     public bool Invalid { get; set; }
 }
```

3.3.2. Inside the same **LapDataModel** class, add the following methods to support mapping with the domain entity:

```
     public Lap MapTo()
     {
         return new Lap()
         {
             Id = Id,
             DriverName = DriverName,
             LapIndex = LapIndex,
             Sector = Sector,
             CreatedOn = CreatedOn,
             ElapsedTime = new TimeSpan(this.ElapsedTime),
             SectorTime = new TimeSpan(this.SectorTime),
             Invalid = Invalid
         };
     }

     public LapDataModel MapFrom(Lap lap)
     {
         Id = lap.Id;
         DriverName = lap.DriverName;
         LapIndex = lap.LapIndex;
```

```
            Sector = lap.Sector;
            ElapsedTime = lap.ElapsedTime.Ticks;
            CreatedOn = lap.CreatedOn;
            Invalid = lap.Invalid;
            return this;
        }
```

3.3.3.Go to the class **SqlLapRepository** (Infrastructure/SqlLap/SqlLapRepository.cs) and the following constructor method:

```
        private readonly string connectionString;

        public SqlLapRepository(string connectionString)
        {
            this.connectionString = connectionString;
        }
```

Setup local connection string. Never store the connection string or any other critical data in a file under source control. In this case, you could override ConnectionStrings:SqlServer managing User Secrets. User secrets could be managed using the following dotnet command:

```
dotnet user-secrets set "ConnectionStrings:SqlServer" "<connection-string>"
```

Visual Sutio also allows to set user secrets right clicking on the API project and selecting the "Manage User Secrets" option.

3.3.4.Add the following to the Add method:

```
        public async Task<Lap> Add(Lap lap)
        {

            using (var connection = new SqlConnection(connectionString))
            {
                await connection.OpenAsync();
                var sqlStatement = @"
                    INSERT INTO [dbo].[Laps]
                               ([Id]
                               ,[DriverName]
                               ,[LapIndex]
                               ,[Sector]
                               ,[ElapsedTime]
                               ,[CreatedOn])
                         VALUES(
                               @Id,
                               @DriverName,
                                @LapIndex,
                                @Sector,
                                @ElapsedTime,
                                @CreatedOn
                           )";
                        await connection.ExecuteAsync(sqlStatement, new 
LapDataModel().MapFrom(lap));
            }
            return lap;
        }
```

3.3.5. Go back to the Post method on Program.cs, start to add a lapRespository parameter to the method (type **ILapRespository**). This enables to get a reference to the repository using Depdency Injection.
Then add the following code at the end of the method:

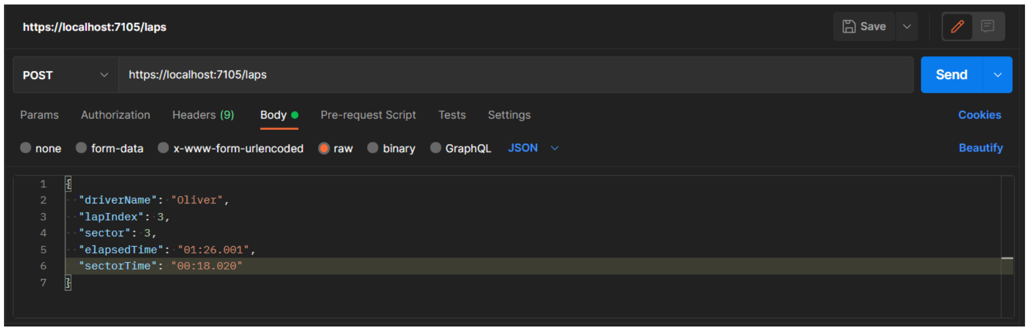
```
        var resultLap = await lapRepository.Add(lap);
```

In this case, we want also to return the new object data as HTTP result. So, we can simply set the resultLap as Results.Ok parameter:

```
        return Results.Ok(resultLap);
```

3.3.6. Because now we are injecting an **ILapRepository** refence in the POST method, we also need to inject a define this instance at the services definition area (Program.cs). Otherwise, the method will try to process this parameter as an expected HTTP content.

3.4. Now we can test the POST method using an HTTP client like Postman:



```
{
  "driverName": "Oliver",
  "lapIndex": 3,
  "sector": 3,
  "elapsedTime": "01:26.001",
  "sectorTime": "00:18.020"
}
```

3.5. Alternativity, we can also call the method using a curl command:

```
curl --location --request POST 'https://localhost:7105/laps' \
--header 'Content-Type: application/json' \
--data-raw '{
  "driverName": "Oliver",
  "lapIndex": 3,
  "sector": 3,
  "elapsedTime": "01:26.001",
  "sectorTime": "00:18.020"
}'
```

4. Get a list of completed lap times.
   4.1. Start to define the GET method on the **SqlLapRepository**:

```
        public async Task<IEnumerable<Lap>> Get()
        {
            using (var connection = new SqlConnection(connectionString))
            {
                await connection.OpenAsync();
                var sqlStatement = @"SELECT * FROM [dbo].[Laps]";
                return (await 
connection.QueryAsync<LapDataModel>(sqlStatement)).Select(l => l.MapTo());
            }
        }
```

   4.2. Now define the Get method with route "/laps" to get a list of lap times:

```
app.MapGet("/laps", async (ILapRepository lapRepository) =>
{
    var results = await lapRepository.Get();
    var finalResults = results.Where(r => r.Sector == 3 && !r.Invalid).OrderBy(r 
=> r.ElapsedTime);
    return Results.Ok(finalResults);
});
```

Only the times with sector equal to 3 (completed laps) and that are valid should be considered.

5. Get a single lap time.
   5.1. Define the repository GetSingle method:

```
      public async Task<Lap> GetSingle(Guid id)
        {
            using (var connection = new SqlConnection(connectionString))
            {
                await connection.OpenAsync();

                var sqlStatement = @"SELECT * FROM [dbo].[Laps] WHERE Id = @Id";
                var foundItems = (await 
connection.QueryAsync<LapDataModel>(sqlStatement, new { Id = id })).Select(l => 
l.MapTo());
                if (foundItems.Any())
                {
                    return foundItems.Single();
                }
                throw new EntityNotFoundException();
            }
        }
```

5.2. Now define the GET method with route “/laps/{id}”, expecting to pass the lap time Id as parameter:

```
app.MapGet("/laps/{id}", async (Guid id, ILapRepository lapRepository) =>
{
    try
    {
        var results = await lapRepository.GetSingle(id);
        return Results.Ok(results);
    }
    catch (EntityNotFoundException ex)
    {
        Console.WriteLine(ex.Message);
        return Results.NotFound();
    }
});
```

6. Update a lap time to be invalid starting to define the Update method on the **SqlLapRepository**:

```
        public async Task Update(Guid id, bool isInvalid)
        {
            using (var connection = new SqlConnection(connectionString))
            {
                await connection.OpenAsync();
                var sqlStatement = @"UPDATE [dbo].[Laps] SET Invalid = @Invalid 
WHERE Id = @Id";
                var rowsAffected = await connection.ExecuteAsync(sqlStatement, 
new { Id = id, Invalid = isInvalid });
                if (rowsAffected == 0)
                {
                    throw new EntityNotFoundException();
                }
            }
        }
```

6.1. Then, the PUT method should be also updated according to the following code:

7. Delete a lap time

7.1. Define the Delete method on the repository side:

```
        public async Task Delete(Guid id)
        {

            using (var connection = new SqlConnection(connectionString))
            {
                await connection.OpenAsync();
                var sqlStatement = @"DELETE FROM [dbo].[Laps] WHERE ID=@Id";
                var rowsAffected = await connection.ExecuteAsync(sqlStatement,
new { Id = id });
                if (rowsAffected == 0)
                {
                    throw new EntityNotFoundException();
                }
            }
        }
```

7.2. Then, create the DELETE method with the route "/laps/{Id}"

```
app.MapDelete("/laps/{id}", async (Guid id, ILapRepository lapRepository) =>
{
    try
    {
        await lapRepository.Delete(id);
        return Results.Ok();
    }
    catch (EntityNotFoundException ex)
    {
        Console.WriteLine(ex.Message);
        return Results.NotFound();
    }
});
```

8. Write an Unit Test to test the GET list method, ensuring that the list is not empty after adding at least one lap to the repository. Include also an Assert to ensure that all the returned laps have always sector=3 by default, even if times for other sectors are inserted before.