

C# Data Access and Entity Framework Core

Rasmus Lystrøm
Associate Professor
ITU
rne@itu.dk

The screenshot displays the Visual Studio Code editor with two C# files open: `ProgramTests.cs` and `Program.cs`. The `ProgramTests.cs` file contains a test class `ProgramTests` with a `[Fact]` attribute and a `Main_given_no_args_p` method. The `Program.cs` file contains a `namespace HelloWorld` with a `public class Program` and a `Main` method that writes "Hello World!" to the console. The terminal window at the bottom shows the output of running the tests, indicating that the test was successful.

```
ProgramTests.cs
1 using System;
2 using System.IO;
3 using Xunit;
4
5 namespace HelloWorld.Tests
6 {
7     0 references | Run All Tests | Debug All Tests
8     public class ProgramTests
9     {
10         [Fact]
11         0 references | Run Test | Debug Test
12         public void Main_given_no_args_p
13         {
14             // Arrange
15             using var writer = new StringWriter();
16             Console.SetOut(writer);
17
18             // Act
19             Program.Main(new string[0]);
20
21             // Assert
22             var output = writer.GetStringBuilder().ToString();
23             Assert.Equal("Hello World!", output);
24         }
25     }
26 }
```

```
Program.cs
1 using System;
2
3 namespace HelloWorld
4 {
5     1 reference
6     public class Program
7     {
8         1 reference
9         public static void Main(string[] args)
10         {
11             Console.WriteLine("Hello World!");
12         }
13     }
14 }
```

TERMINAL

```
1: pwsh
Loading personal and system profiles took 1008ms.
C:\HelloWorld> dotnet test
Test run for C:\HelloWorld\HelloWorld.Tests\bin\Debug\netcoreapp3.0\HelloWorld.Tests.dll (.NETCoreApp, Version=v3.0)
Microsoft (R) Test Execution Command Line Tool Version 16.3.0
Copyright (c) Microsoft Corporation. All rights reserved.

Starting test execution, please wait...

A total of 1 test files matched the specified pattern.

Test Run Successful.
Total tests: 1
Passed: 1
Total time: 3,4618 Seconds
C:\HelloWorld>
```

Agenda

Databases (SQL Server)

Secrets

Old school SQL

SQL Injection

The IDisposable interface

Entity Framework Core

Lazy vs. Eager Loading

Databases



Databases

Relational (SQL)

Microsoft SQL Server

Oracle Database

IBM Db2

MySQL

MariaDB

PostgreSQL

SQLite

Document (NoSQL)

Azure Cosmos DB

Amazon DynamoDB

MongoDB

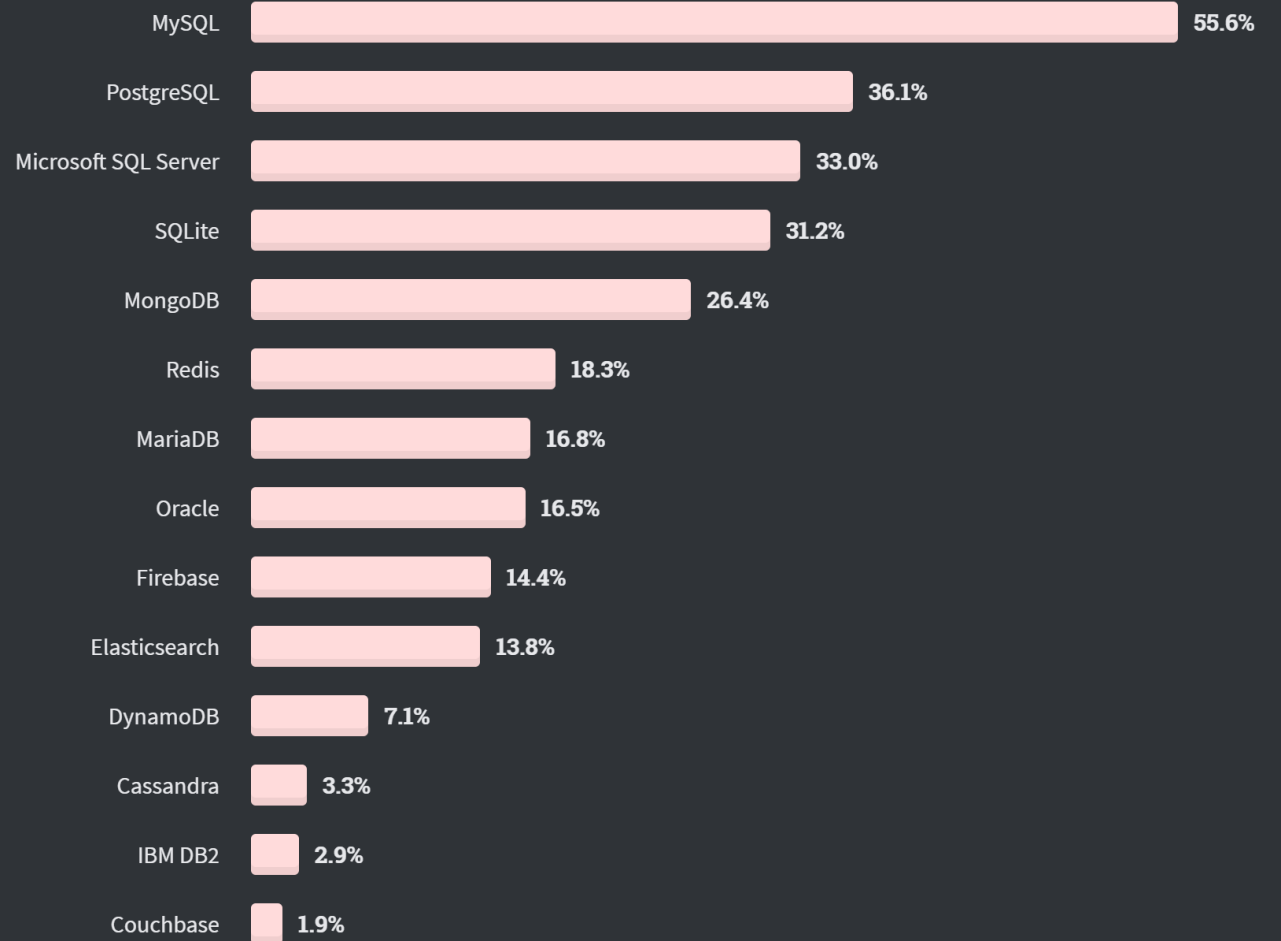
Couchbase

Redis

Elasticsearch

Most popular databases

<https://insights.stackoverflow.com/survey/2020#most-popular-technologies>



SQL Server

Windows only: SQL Server Express LocalDB

<https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/sql-server-express-localdb>

All (preferred): SQL Server in a container

Get docker:

<https://docs.docker.com/get-docker/>

Run container:

<https://docs.microsoft.com/en-us/sql/linux/quickstart-install-connect-docker>

SQL Server

```
docker pull mcr.microsoft.com/mssql/server:2019-latest
```

```
$password = New-Guid
```

```
docker run -e "ACCEPT_EULA=Y" -e "SA_PASSWORD=$password" `
  -p 1433:1433 `
  -d mcr.microsoft.com/mssql/server:2019-latest
```




SQL Server Demo

SQL Server Docker Container



Secrets



Secrets

```
dotnet user-secrets init
```

```
dotnet user-secrets set "ConnectionStrings:ConnectionString" "..."
```

```
dotnet add package Microsoft.Extensions.Configuration.UserSecrets
```

Secrets

```
using Microsoft.Extensions.Configuration;
```

```
...
```

```
var configuration = new ConfigurationBuilder()  
    .AddUserSecrets(typeof(Program).Assembly)  
    .Build();
```

```
var connectionString = configuration.GetConnectionString("ConnectionString");
```

Old School SQL

Old School SQL

```
dotnet add package System.Data.SqlClient
```

```
var cmdText = "SELECT * FROM Animals";  
using var connection = new SqlConnection(connectionString);  
connection.Open();  
using var command = new SqlCommand(cmdText, connection);  
using var reader = command.ExecuteReader();  
while (reader.Read())  
{  
    ...  
}
```

SQL Injection

SQL Injection

A **SQL injection** attack consists of insertion or “injection” of a SQL query via the input data from the client to the application.

A successful SQL injection exploit can:

- read sensitive data from the database,
- modify database data (Insert/Update/Delete),
- execute administration operations on the database (such as shutdown the DBMS),
- or worse

IDisposable

IDisposable

```
try
{
    ...
}
finally
{
    resource.Dispose();
}
```

IDisposable

```
using (var resource1 = new Resource())  
{  
    ...  
}
```

IDisposable

```
using var resource = new Resource();
```

```
...
```

Entity Framework Core

Entity Framework Core

```
dotnet tool install --global dotnet-ef
```

```
dotnet add package Microsoft.EntityFrameworkCore.Design
```

```
dotnet ef migrations add InitialCreate
```

```
dotnet ef database update
```

Lazy Loading

<https://docs.microsoft.com/en-us/ef/core/querying/related-data/lazy>

Lazy Loading

```
dotnet add package Microsoft.EntityFrameworkCore.Proxies
```

```
protected override void OnConfiguring2(DbContextOptionsBuilder optionsBuilder)  
    => optionsBuilder.UseLazyLoadingProxies()  
        .UseSqlServer(...);
```