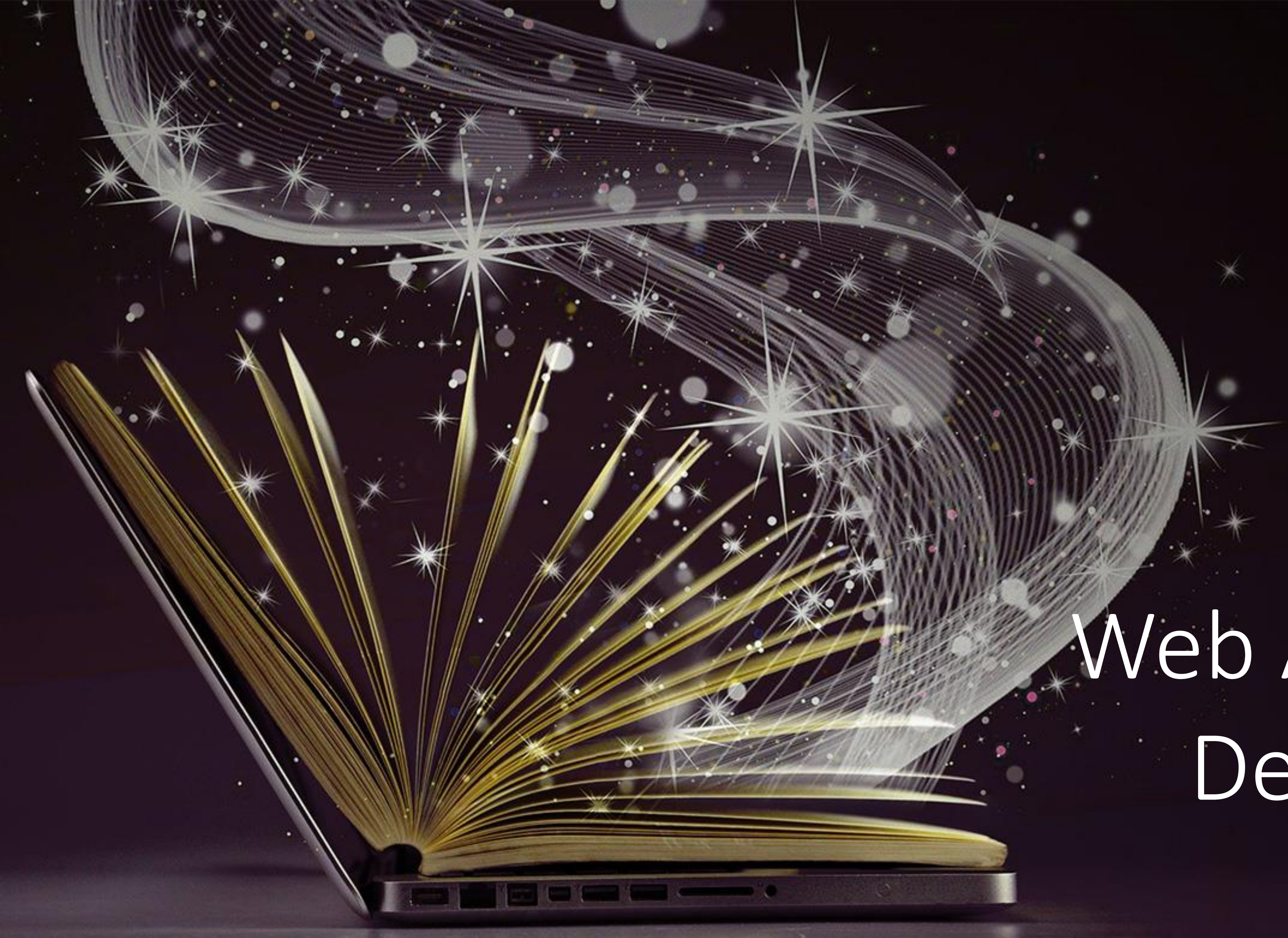


# CENG 423 Web Application Development

Working with Data – Part II

# Announcements

- I plan to make changes on the grading policy
- Currently
  - Midterm 20%
  - Final 30%
  - Project 30%
  - Assignment 20%

- The new scheme
  - Midterm 20%
  - Final 60%
  - Assignment 20%

# Announcements

- The new scheme
  - Midterm 20%
  - Final 60%
  - Assignment 20%

- Basically, the final exam will be in the form of project/assignment
- The project due date will be the day of the final exam
- During the final exam time, everyone will be on the Teams and present their project
- Everyone should take the equal amount of speaking time
- You are required to use a version control system such as GitHub do individual commits
- I will check the individual contributions and grade students based on their contribution (writing a report is not a contribution)

# Changes on The Project

- Since the project becomes the final exam, no two groups will work on the same topic
- You are required to write a proposal explaining your scope
- The scope can be including but not limited to followings
  - Online bookstore
    - Book catalog, inventory, and shopping features
  - Movie database
    - Movie catalog, user comments, and rating features
  - Restaurant delivery service
    - Restaurants catalogs, restaurants menu, ordering, and rating features
  - Online used stuff store
    - Stuff categories, conditions, and shopping features
  - Hotel reservation
    - Hotel catalogs, amenities, rates, and shopping features

# Changes on The Project

- The proposal is due June 4th Sunday, at 23.59
- Late submission will be penalized 10% of the final grade for every 24 hours
- One submission for each group will suffice
  - Please do not forget to add the name of all teammates and the group number
- If two or more groups will pick the same subject, I will assign random topics to groups so that no two groups will work on the same subject

# Changes on The Project

- The proposal must consist of the following sections:
  - Introduction:
    - Mention the significance of the project proposal and how it aligns with the course objectives
  - Project Description:
    - Describe the proposed web-based application project in detail
    - Explain the problem or challenge that the application aims to address or the specific functionality it aims to provide
    - Identify the target audience or users who will benefit from the application
    - Discuss the potential impact or benefits of the proposed application
  - Project Objectives:
    - Clearly state the specific objectives of the project
    - List the main features and functionalities that the application will include

# ASP.NET Core Data Features

- Data features allow responses to be produced using data that has been previously created by an earlier request or stored in a database
- Data features are useful as
  - most web applications deal with data that is expensive to re-create for every request
  - allow responses to be produced more efficiently and with fewer resources
- Data features are used as
  - Data values are cached using a service
  - Responses are cached by a middleware component based on the **Cache-Control** header
  - Databases are accessed through a service that translates LINQ (language-integrated query) into SQL statements
  - LINQ is a powerful set of technologies based on the integration of query capabilities directly into the C# language
- They are optional

# Using Entity Framework Core

- Not all data values are produced directly by the application, and most projects will need to access data in a database
- Entity Framework Core is well-integrated into the ASP.NET Core platform, with good support for creating a database from C# classes and for creating C# classes to represent an existing database

# Installing Entity Framework Core

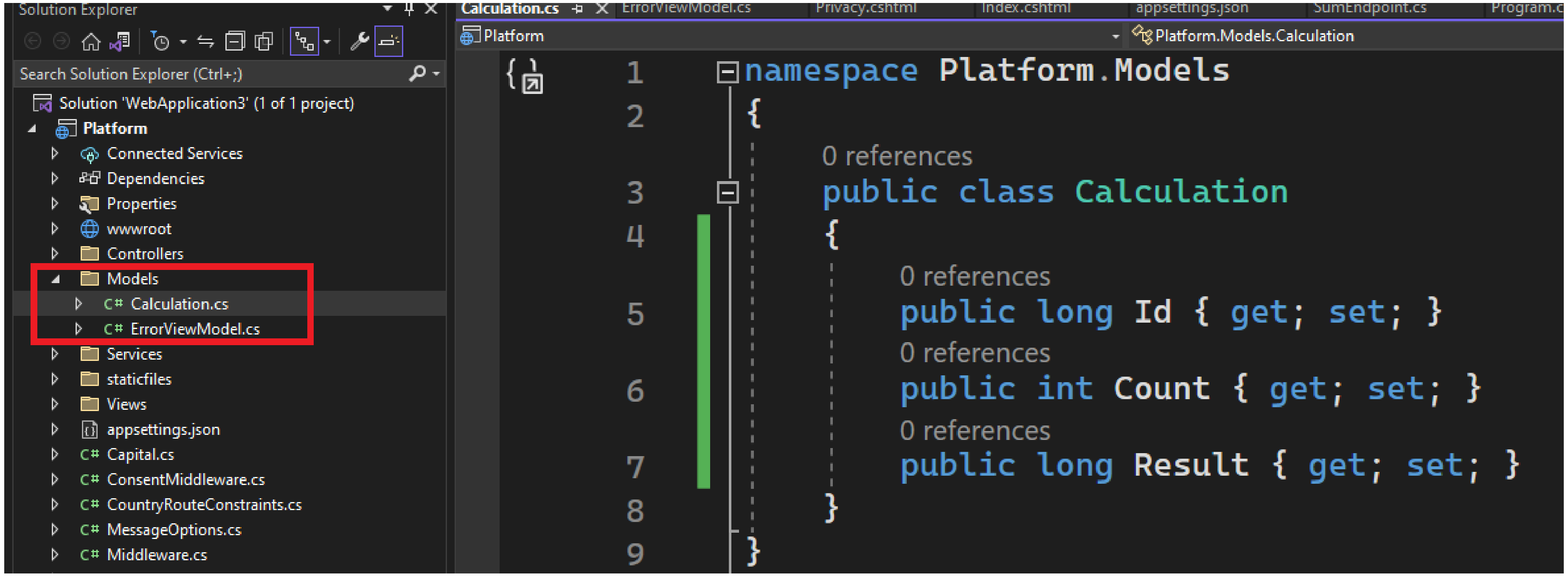
- Entity Framework Core requires a global tool package that is used to manage databases from the command line and to manage packages for the project that provide data access
- To install the tools package, open the terminal and run the following commands
  - dotnet tool uninstall --global dotnet-ef
  - dotnet tool install --global dotnet-ef --version 6.0.0
- The first line removes any existing version of dotnet-ef package
- The second line installs the Entity Framework version we need
- Final, run the following commands to install Entity Framework Core packages
  - dotnet add package Microsoft.EntityFrameworkCore.Design --version 6.0.0
  - dotnet add package Microsoft.EntityFrameworkCore.SqlServer --version 6.0.0

# Creating the Data Model

- The Model is the part of MVC which implements the domain logic
- This logic is used to handle the data passed between the database and the user interface
- To create the model, we are going to use the Models folder

# Creating the Data Model

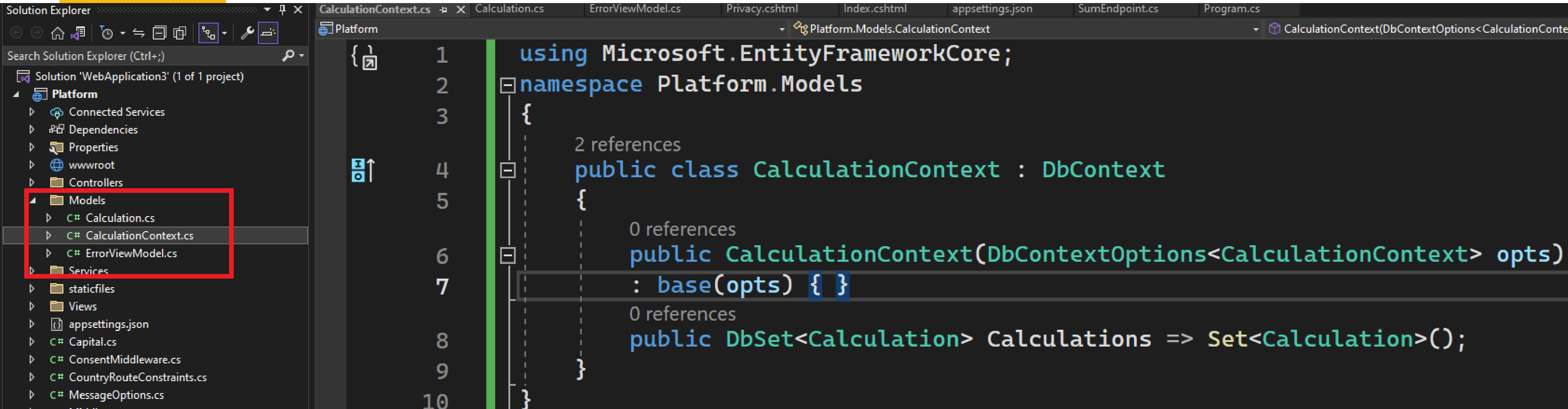
- Create Calculation.cs file under the Models folder
- The Id property will be used to create a unique key for each object stored in the database
- The Count and Result properties will describe a calculation and its result
- Now we need a class providing the access to the database



```
namespace Platform.Models
{
    public class Calculation
    {
        public long Id { get; set; }
        public int Count { get; set; }
        public long Result { get; set; }
    }
}
```

# Creating the Data Model

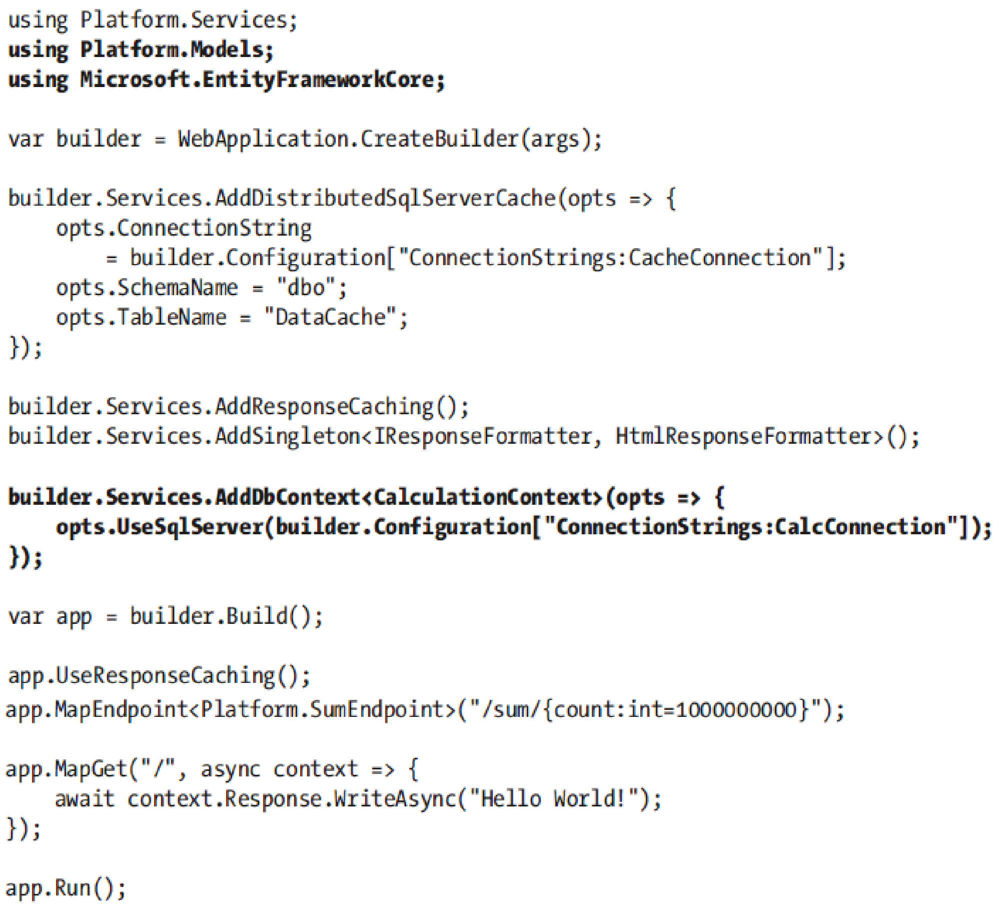
- Entity Framework Core uses a context class that provides access to the database
- Create CalculationContext.cs file under the Models folder
- This class defines a constructor that is used to receive an options object that is passed on to the base constructor
- The Calculations property provides access to the Calculation objects that Entity Framework Core will retrieve from the database



```
using Microsoft.EntityFrameworkCore;
namespace Platform.Models
{
    public class CalculationContext : DbContext
    {
        public CalculationContext(DbContextOptions<CalculationContext> opts)
        : base(opts) { }
        public DbSet<Calculation> Calculations => Set<Calculation>();
    }
}
```

# Configuring the Database Service

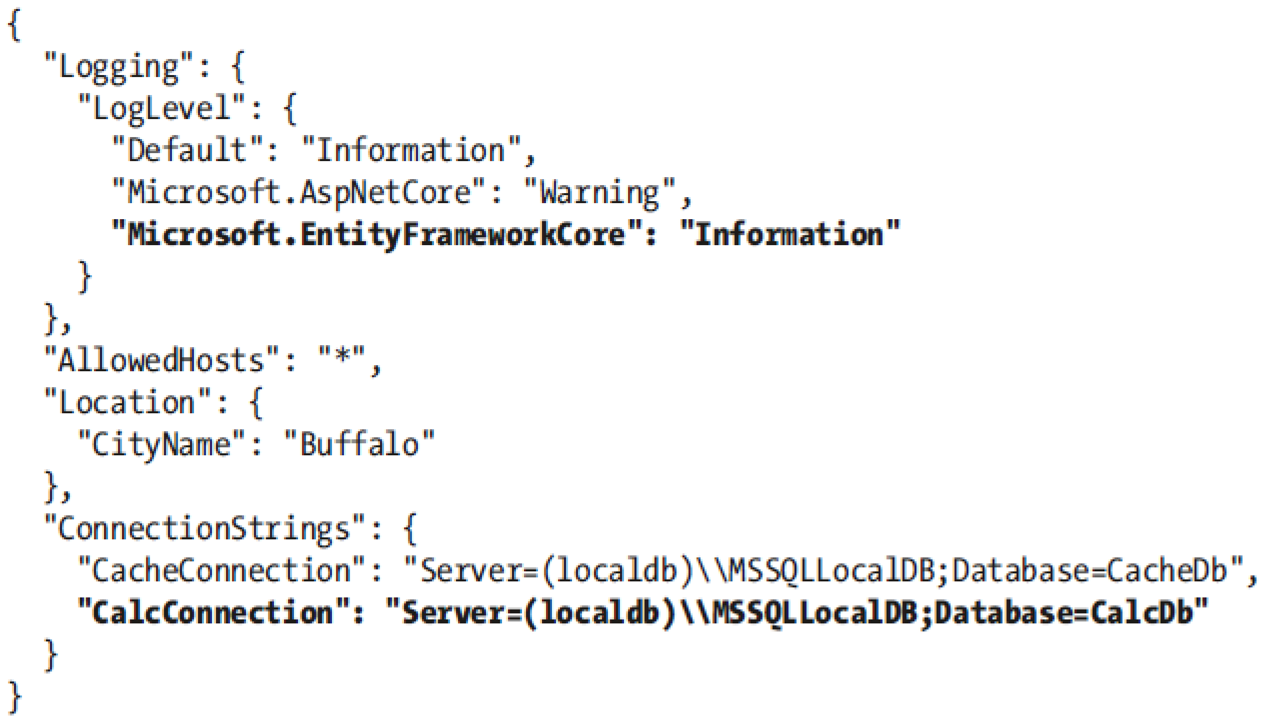
- Access to the database is provided through a service
- The AddDbContext method creates a service for an Entity Framework Core context class
- It receives an options object that is used to select the database provider, which is done with the UseSqlServer method



```
using Platform.Services;
using Platform.Models;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDistributedSqlServerCache(opts => {
    opts.ConnectionString
        = builder.Configuration["ConnectionStrings:CacheConnection"];
    opts.SchemaName = "dbo";
    opts.TableName = "DataCache";
});

builder.Services.AddResponseCaching();
builder.Services.AddSingleton<IResponseFormatter, HtmlResponseFormatter>();

builder.Services.AddDbContext<CalculationContext>(opts => {
    opts.UseSqlServer(builder.Configuration["ConnectionStrings:CalcConnection"]);
});

var app = builder.Build();

app.UseResponseCaching();
app.MapEndpoint<Platform.SumEndpoint>("/sum/{count:int=1000000000}");

app.MapGet("/", async context => {
    await context.Response.WriteAsync("Hello World!");
});

app.Run();
```

# Configuring the Database Service

- The IConfiguration service is used to get the connection string for the database
- To define the connection string do the following changes in the appsettings.json File



```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.EntityFrameworkCore": "Information"
    }
  },
  "AllowedHosts": "*",
  "Location": {
    "CityName": "Buffalo"
  },
  "ConnectionStrings": {
    "CacheConnection": "Server=(localdb)\\MSSQLLocalDB;Database=CacheDb",
    "CalcConnection": "Server=(localdb)\\MSSQLLocalDB;Database=CalcDb"
  }
}
```

# Creating and Applying the Database Migration

- Entity Framework Core manages the relationship between data model classes and the database using a feature called ***migrations***
- When changes are made to the model classes, a new migration is created that modifies the database to match those changes
- To create the initial migration, which will create a new database and prepare it to store Calculation objects, open a new PowerShell command prompt, navigate to the folder that contains the Platform.csproj file, and run the command below:
- dotnet ef migrations add Initial

# Creating and Applying the Database Migration

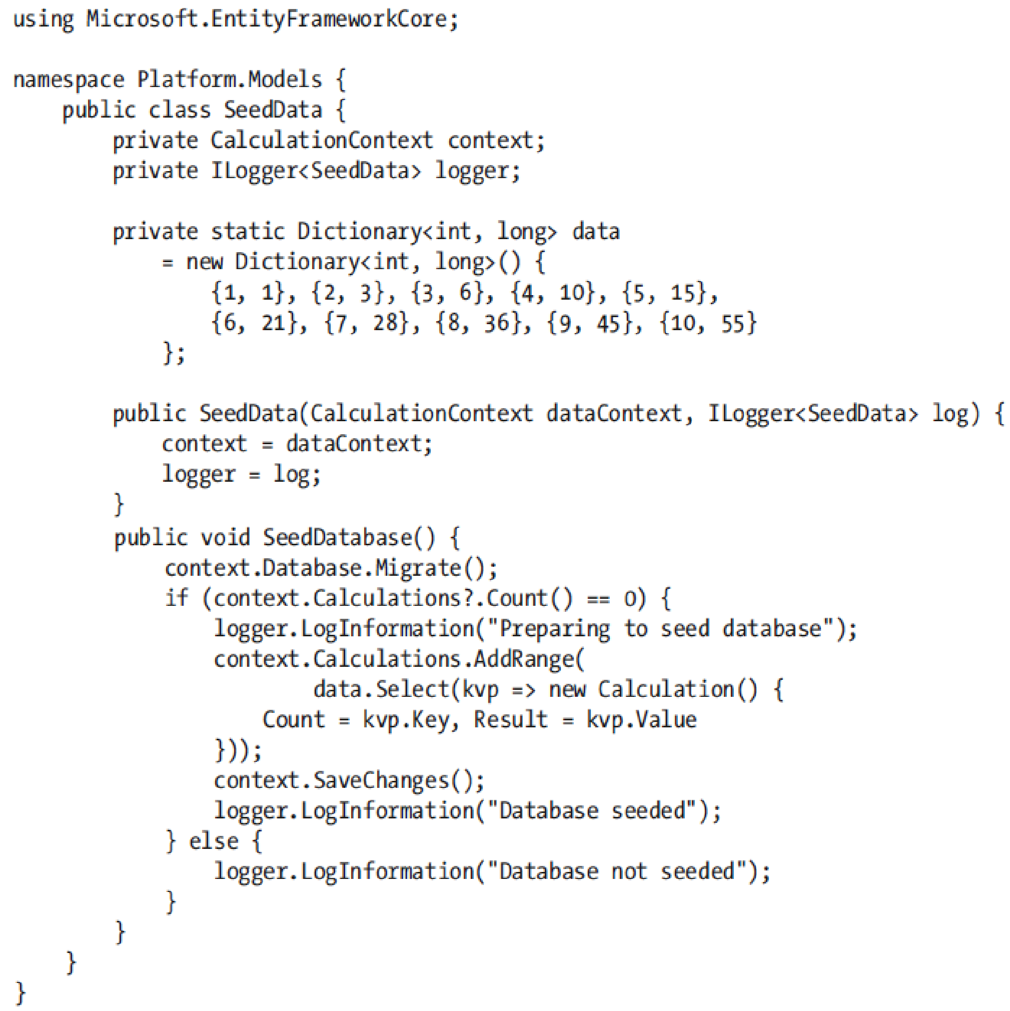
- The dotnet ef commands relate to Entity Framework Core
- The command creates a new migration named ***Initial***, which is the name conventionally given to the first migration for a project
- You will see that a Migrations folder has been added to the project and that it contains class files whose statements prepare the database so that it can store the objects in the data model
- To apply the migration, run the command below in the Platform project folder
- dotnet ef database update
- This command executes the commands in the migration created and uses them to prepare the database, which you can see in the SQL statements written to the command prompt

# Seeding the Database

- Most applications require some seed data, especially during development
- Entity Framework Core does provide a database seeding feature, but it is of limited use for most projects
  - it doesn’t allow data to be seeded where the database allocates unique keys to the objects it stores
- This is an important feature in most data models because it means the application doesn’t have to worry about allocating unique key values
- More flexible approach is to use the regular Entity Framework Core features to add seed data to the database

# Seeding the Database

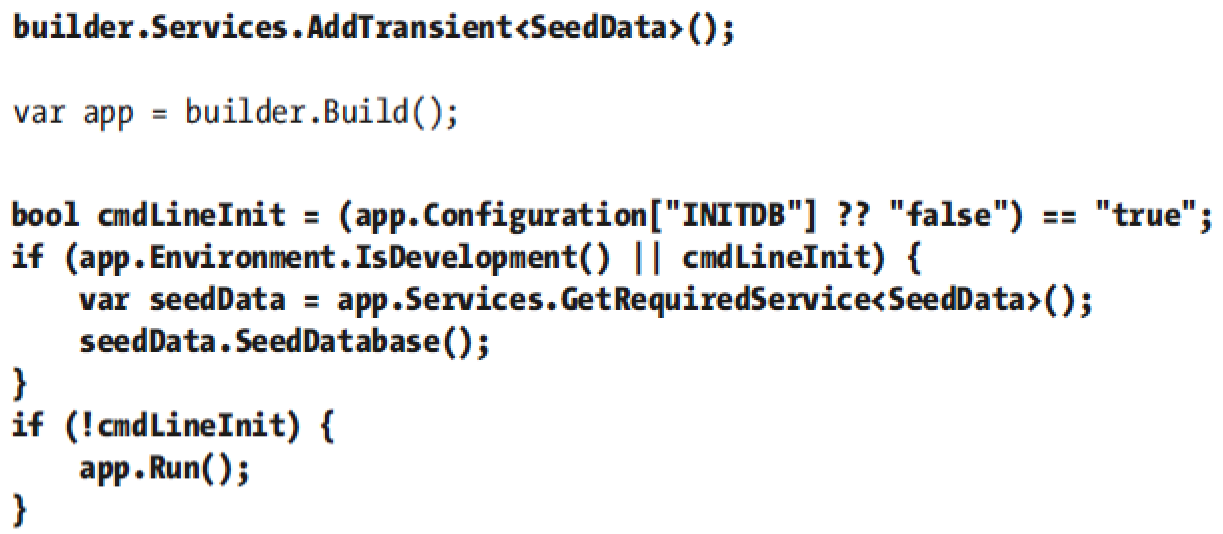
- Create file called SeedData.cs in the Platform/Models folder
- The SeedData class declares constructor dependencies on the CalculationContext and ILogger<T> types, which are used in the SeedDatabase method to prepare the database
- Database.Migrate method is used to apply any pending migrations to the database
- Calculations property is used to store new data using the AddRange method
- The new objects are stored in the database using the SaveChanges method



```
using Microsoft.EntityFrameworkCore;

namespace Platform.Models {
    public class SeedData {
        private CalculationContext context;
        private ILogger<SeedData> logger;

        private static Dictionary<int, long> data
            = new Dictionary<int, long>() {
                {1, 1}, {2, 3}, {3, 6}, {4, 10}, {5, 15},
                {6, 21}, {7, 28}, {8, 36}, {9, 45}, {10, 55}
            };

        public SeedData(CalculationContext dataContext, ILogger<SeedData> log) {
            context = dataContext;
            logger = log;
        }
        public void SeedDatabase() {
            context.Database.Migrate();
            if (context.Calculations?.Count() == 0) {
                logger.LogInformation("Preparing to seed database");
                context.Calculations.AddRange(
                        data.Select(kvp => new Calculation() {
                    Count = kvp.Key, Result = kvp.Value
                }));
                context.SaveChanges();
                logger.LogInformation("Database seeded");
            } else {
                logger.LogInformation("Database not seeded");
            }
        }
    }
}
```

# Seeding the Database

- To use the SeedData class, make the changes below in the Program.cs file



```
builder.Services.AddTransient<SeedData>();

var app = builder.Build();

bool cmdLineInit = (app.Configuration["INITDB"] ?? "false") == "true";
if (app.Environment.IsDevelopment() || cmdLineInit) {
    var seedData = app.Services.GetRequiredService<SeedData>();
    seedData.SeedDatabase();
}
if (!cmdLineInit) {
    app.Run();
}
```

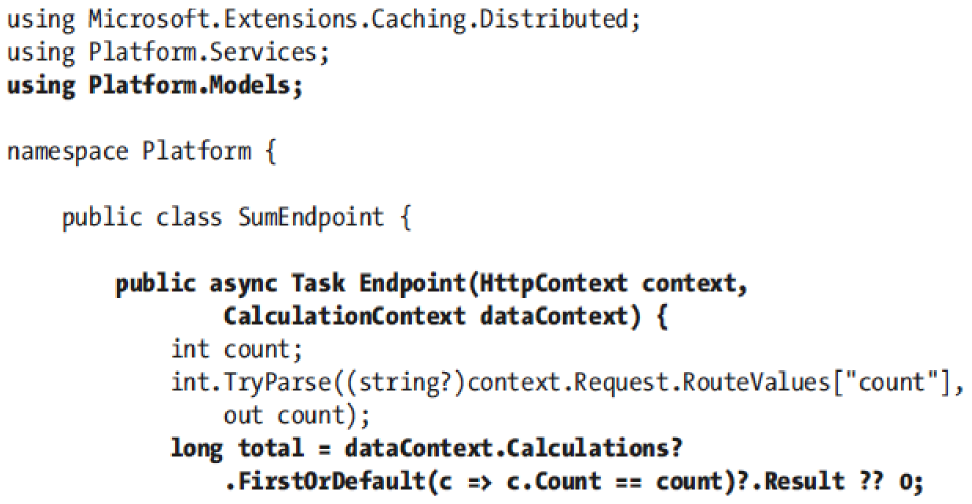
- We build the service for the SeedData class so that it will be instantiated, its dependencies will be resolved
- To seed the database, we need to run the following command
- dotnet run INITDB=true

# Seeding the Database

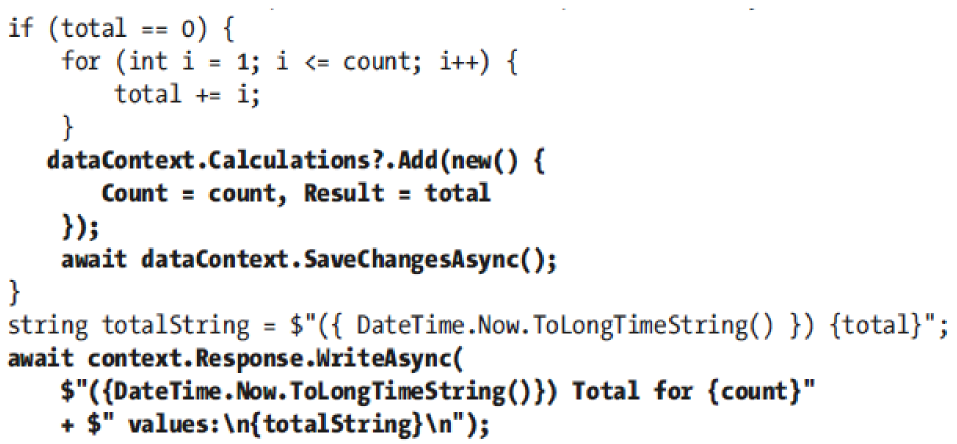
- We build the service for the SeedData class so that it will be instantiated, its dependencies will be resolved
- To seed the database, we need to run the following command
- dotnet run INITDB=true
- The application will start, and the database will be seeded with the results for the ten calculations defined by the SeedData class, after which the application will terminate
- During the seeding process the SQL statements that are sent to the database are shown
  - Check to see whether there are any pending migrations
  - Count the number of rows in the table used to store Calculation data
  - If the table is empty, add the seed data

# Using Data in an Endpoint

- Endpoints and middleware components access Entity Framework Core data by declaring a dependency on the context class and using its DbSet<T> properties to perform LINQ queries
- The LINQ queries are translated into SQL and sent to the database
- The row data received from the database is used to create data model objects that are used to produce responses
- The update on the SumEndpoint class allows us to use Entity Framework Core



```
using Microsoft.Extensions.Caching.Distributed;
using Platform.Services;
using Platform.Models;

namespace Platform {

    public class SumEndpoint {

        public async Task Endpoint(HttpContext context,
                CalculationContext dataContext) {
            int count;
            int.TryParse((string?)context.Request.RouteValues["count"],
                out count);
            long total = dataContext.Calculations?
                .FirstOrDefault(c => c.Count == count)?.Result ?? 0;
```



```
if (total == 0) {
    for (int i = 1; i <= count; i++) {
        total += i;
    }
   dataContext.Calculations?.Add(new() {
       Count = count, Result = total
    });
    await dataContext.SaveChangesAsync();
}
string totalString = $"({ DateTime.Now.ToLongTimeString() }) {total}";
await context.Response.WriteAsync(
    $"({DateTime.Now.ToLongTimeString()}) Total for {count}"
    + $" values:\n{totalString}\n");
```

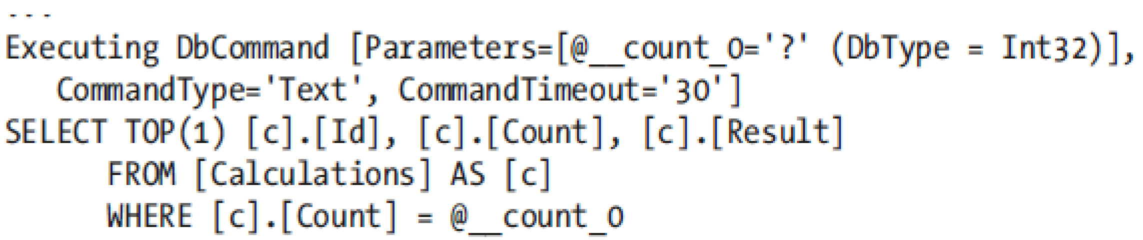
# Using Data in an Endpoint

- The endpoint uses the LINQ FirstOrDefault to search for a stored Calculation object for the calculation that has been requested like this
  - dataContext.Calculations?.FirstOrDefault(c => c.Count == count)?.Result ?? 0;

- If an object has been stored, it is used to prepare the response
- If not, then the calculation is performed, and a new Calculation object is stored by these statements:
  - dataContext.Calculations?.Add(new Calculaton() { Count = count, Result = total});
  - await dataContext.SaveChangesAsync();

# Using Data in an Endpoint

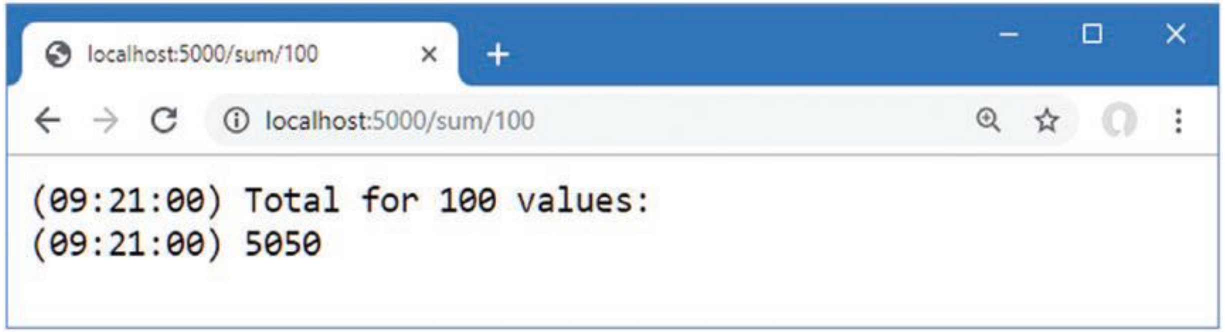
- The Add method is used to tell Entity Framework Core that the object should be stored, but the update isn’t performed until the SaveChangesAsync method is called
- To see that restart the ASP and visit http://localhost:5000/sum/10
- This is one of the calculations with which the database has been seeded, and you will be able to see the query sent to the database in the logging messages produced by the application



```
...
Executing DbCommand [Parameters=[@__count_0='?' (DbType = Int32)],
   CommandType='Text', CommandTimeout='30']
SELECT TOP(1) [c].[Id], [c].[Count], [c].[Result]
      FROM [Calculations] AS [c]
      WHERE [c].[Count] = @__count_0
```

# Using Data in an Endpoint

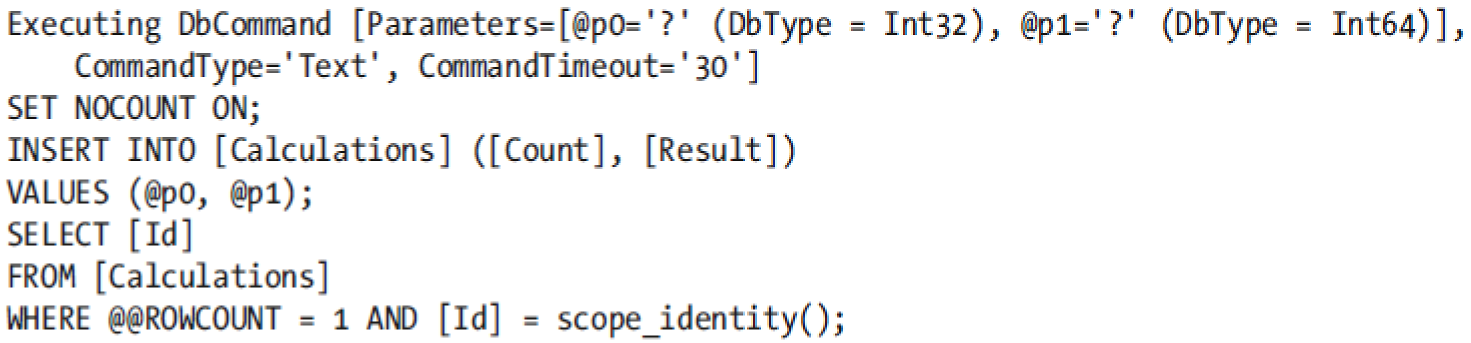
- If you request http://localhost:5000/sum/100, the database will be queried, but no result will be found
- The endpoint performs the calculation and stores the result in the database before producing the result



```
(09:21:00) Total for 100 values:
(09:21:00) 5050
```

# Using Data in an Endpoint

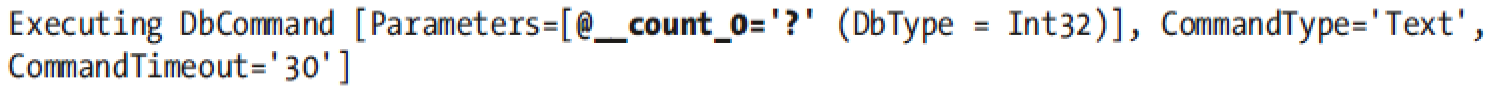
- Once a result has been stored in the database, subsequent requests for the same URL will be satisfied using the stored data
- The SQL statement used to store the data in the logging output produced by Entity Framework Core is shown in Terminal



```
Executing DbCommand [Parameters=[@p0='?' (DbType = Int32), @p1='?' (DbType = Int64)],
    CommandType='Text', CommandTimeout='30']
SET NOCOUNT ON;
INSERT INTO [Calculations] ([Count], [Result])
VALUES (@p0, @p1);
SELECT [Id]
FROM [Calculations]
WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();
```

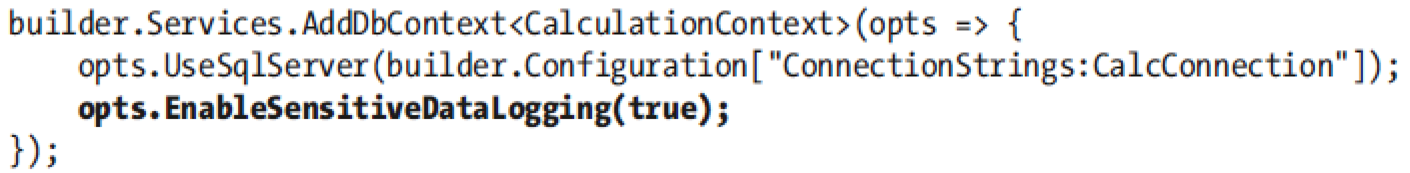
# Enabling Sensitive Data Logging

- Entity Framework Core doesn’t include parameter values in the logging messages it produces, which is why the logging output contains question marks, like this



```
Executing DbCommand [Parameters=[@__count_0='?' (DbType = Int32)], CommandType='Text',
CommandTimeout='30']
```

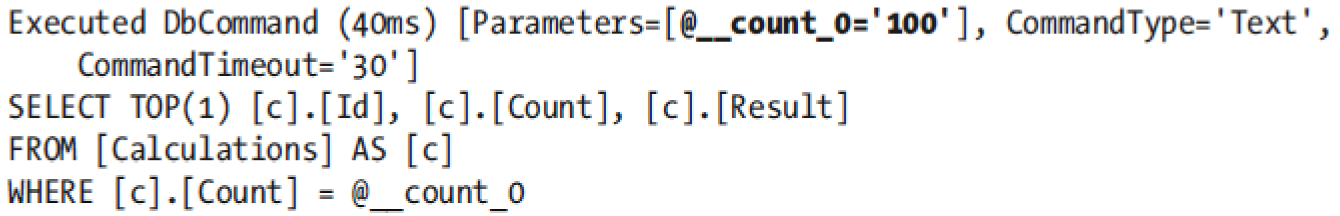
- The data is omitted as a precaution to prevent sensitive data from being stored in logs
- If you are having problems with queries and need to see the values sent to the database, then you can use the EnableSensitiveDataLogging method when configuring the database context



```
builder.Services.AddDbContext<CalculationContext>(opts => {
    opts.UseSqlServer(builder.Configuration["ConnectionStrings:CalcConnection"]);
    opts.EnableSensitiveDataLogging(true);
});
```

# Enabling Sensitive Data Logging

- Restart ASP.NET Core MVC and request the http://localhost:5000/sum/100 URL again
- When the request is handled, Entity Framework Core will include parameter values in the logging message it creates to show the SQL query, like this



```
Executed DbCommand (40ms) [Parameters=[@__count_0='100'], CommandType='Text',
    CommandTimeout='30']
SELECT TOP(1) [c].[Id], [c].[Count], [c].[Result]
FROM [Calculations] AS [c]
WHERE [c].[Count] = @__count_0
```

- This is a feature that should be used with caution because logs are often accessible by people who would not usually have access to the sensitive data that applications handle such as credit card numbers and account details