Working with Databases

Database in Distributed computing

- In distributed computing, databases play a crucial role in managing and storing data across multiple nodes or servers in a network.
- Distributed databases are designed to provide scalability, fault tolerance, and improved performance compared to centralized databases.
- Data Distribution: In a distributed database system, data is distributed across multiple nodes or servers. This distribution can be done in various ways, such as sharding, partitioning, or replication. The goal is to ensure that data is available and accessible even if some nodes fail.

Distributed Database

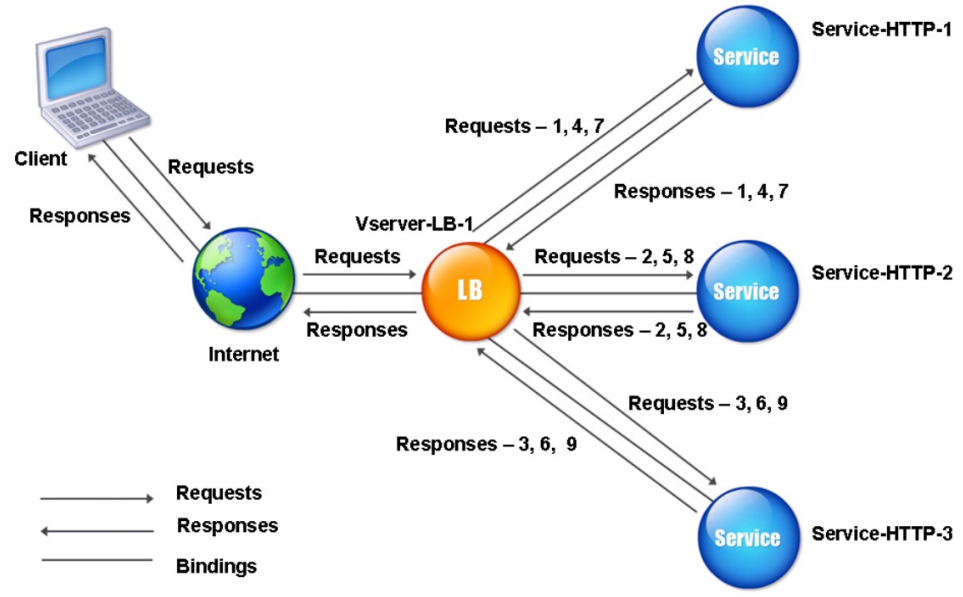
- Replication: Replicating data involves maintaining multiple copies of the same data on different nodes. This provides fault tolerance and high availability. Changes to one copy of the data are propagated to other copies to keep them synchronized. However, managing consistency and avoiding conflicts is a challenge in replication-based systems.
- Scalability: One of the primary reasons for using distributed databases is scalability. As data grows, you can add more nodes to the database cluster to distribute the load and maintain acceptable performance levels. Horizontal scaling, where you add more machines, is a common approach.

Distributed database (Cont..)

- Security and Authorization: Distributed databases must implement robust security measures to protect data from unauthorized access. Access control, encryption, and auditing mechanisms are essential components of distributed database security.
- Data Consistency and Conflict Resolution: In a distributed environment, conflicts can occur when multiple nodes attempt to update the same data simultaneously. Conflict resolution mechanisms must be in place to handle such situations and maintain data integrity.

Distributed database (Cont..)

- Load Balancing: To ensure even distribution of query and transaction loads across nodes, load balancing mechanisms are employed. Load balancers distribute incoming requests to database nodes based on various algorithms like round-robin, least connections, or weighted distribution.



Round Robin load balancer – source: https://docs.netscaler.com/en-us/citrix-adc/current-release/load-balancing/load-balancing-customizing-algorithms/roundrobin-method.html

Commercial database servers

- Commercial database servers are software products provided by various companies for managing and storing structured data efficiently and securely. Here are some well-known commercial database servers:
  - **Oracle Database**
  - **Microsoft SQL Server**
  - **Amazon RDS (Relational Database Service)**
  - **Google Cloud SQL**
  - **Microsoft Azure SQL Database**

Portable Database

- A portable database refers to a database system that can be easily moved or transferred from one computing environment to another with minimal effort or modification.
- Portability in the context of databases is important for scenarios where data needs to be moved between different systems, platforms, or locations without significant changes to the database structure or code.
- Open source database systems like MySQL, PostgreSQL, and SQLite are known for their portability. They have versions available for various operating systems and architectures, making it easier to transfer data between systems.
- Technologies like virtual machines (VMs) and containerization (e.g., Docker) can encapsulate an entire database system, including its dependencies and configuration, into a portable package. This package can be moved to different environments without modification.

Sqlite database

- Self-Contained: SQLite is a serverless, self-contained database engine, meaning it doesn't require a separate server process to operate. The entire database is contained within a single file on disk, making it highly portable and easy to distribute.
- File-Based: SQLite databases are stored as single files on the file system. These files have a ".db" or ".sqlite" extension and contain all the necessary data and schema information.
- Cross-Platform: SQLite is cross-platform and runs on various operating systems, including Windows, macOS, Linux, Android, and iOS.

Why are we learning Sqlite

- SQLite is commonly used for prototyping and development due to its ease of use and portability.
- However, for high-concurrency or large-scale applications, you may want to consider more robust RDBMS solutions like MySQL, PostgreSQL, or Microsoft SQL Server, which offer features like multi-user support and advanced management capabilities.
- Our main focus is the **universal database connectivity.**
- A universal approach to database connectivity aims to provide a standardized way to connect to different types of database management systems (DBMS) regardless of the programming language or platform you are using.

Universal database connectivity strategy

- **ODBC (Open Database Connectivity):** ODBC is a standard API for connecting to databases. It provides a common interface for various programming languages, including C/C++, Python, Java, and more, to access different DBMSs. ODBC drivers are available for many popular databases, making it a universal choice for cross-DBMS connectivity.
- **JDBC (Java Database Connectivity):** JDBC is the Java equivalent of ODBC and is specifically designed for Java applications. It offers a standardized way to connect to and interact with relational databases using Java.
- **ADO.NET:** For .NET-based applications, ADO.NET provides a set of classes and libraries for database connectivity. It supports various database providers through Data Providers, which are specific to the target DBMS.

Database basics - Table

- A table is a collection of related data organized into rows and columns. Each row represents a record, and each column represents a specific attribute of the data.
- Example: Table name: Employees

EmployeeID	FirstName	LastName	Department	Salary
1	John	Smith	HR	$60000
2	Jane	Doe	Marketing	$55000
3	Michael	Johnson	Sales	$70000
4	Emily	Williams	IT	$65000

Row

Column

Database basics – Primary key

- A primary key is a unique identifier for each row in a table. It ensures that each record is uniquely identified and helps maintain data integrity.

Primary key

EmployeeID	FirstName	LastName	Department	Salary
1	John	Smith	HR	$60000
2	Jane	Doe	Marketing	$55000
3	Michael	Johnson	Sales	$70000
4	Emily	Williams	IT	$65000

Database basic – Foreign Key

- Foreign Key: A foreign key is a column in one table that references the primary key in another table. It establishes relationships between tables and ensures data consistency.
- Let's extend the previous "Employees" example to include a second table that demonstrates the use of a foreign key relationship. In this example, we'll add a "Departments" table and establish a relationship between the two tables using a foreign key.

Database basic – Foreign Key

Table Name: Departments

DepartmentID	DepartmentName
1	HR
2	Marketing
3	Sales
4	IT

DepartmentID – Primary key

Table Name: Employees

EmployeeID	FirstName	LastName	DepartmentID	Salary
1	John	Smith	1	$60000
2	Jane	Doe	2	$55000
3	Michael	Johnson	3	$70000
4	Emily	Williams	4	$65000

EmployeeID – Primary Key

DepartmentID – foreign key

SQLite syntax – Create table

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
    ...
);
```

- create a table named "Students" with columns for "id" and "name”

```
CREATE TABLE Students (
    id INTEGER,
    name TEXT
);
```

```
CREATE TABLE Students (
    id INTEGER PRIMARY KEY,
    name TEXT
);
```

With Primary key

```
CREATE TABLE Students (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT
);
```

With Auto increment

SQLite syntax - INSERT

- To insert data into an SQLite table, you use the INSERT INTO statement. Here's how you would insert a new row of data into a table named "Students" with "id" and "name" columns

```
INSERT INTO Students (id, name)
VALUES (1, 'John Doe');
```

- If id is auto generated, you can omit it:

```
INSERT INTO Students (name)
VALUES ('Jane Smith');
```

SQLite syntax - Update

- Suppose you have the following initial data:

id	name
1	John Doe
2	Jane Smith

- And you want to update the name of the student with "id" 2 to "Jane Johnson". The SQL UPDATE statement would look like this:

```
UPDATE Students
SET name = 'Jane Johnson'
WHERE id = 2;
```

SQLite syntax - Delete

- Delete all rows:

```
DELETE FROM Students
```

- Delete a specific row:

```
DELETE FROM Students
WHERE id = 2;
```

SQLite syntax - Select

- Select statement allows you to query the database and retrieve specific rows and columns of data based on your criteria.
- Suppose you have a table named "Students" with the following data:

id	name
1	John Doe
2	Jane Smith
3	Alice Brown

- Select all data:

```
SELECT * FROM Students;
```

- Select a specific column:

```
SELECT name FROM Students;
```

- Select with condition:

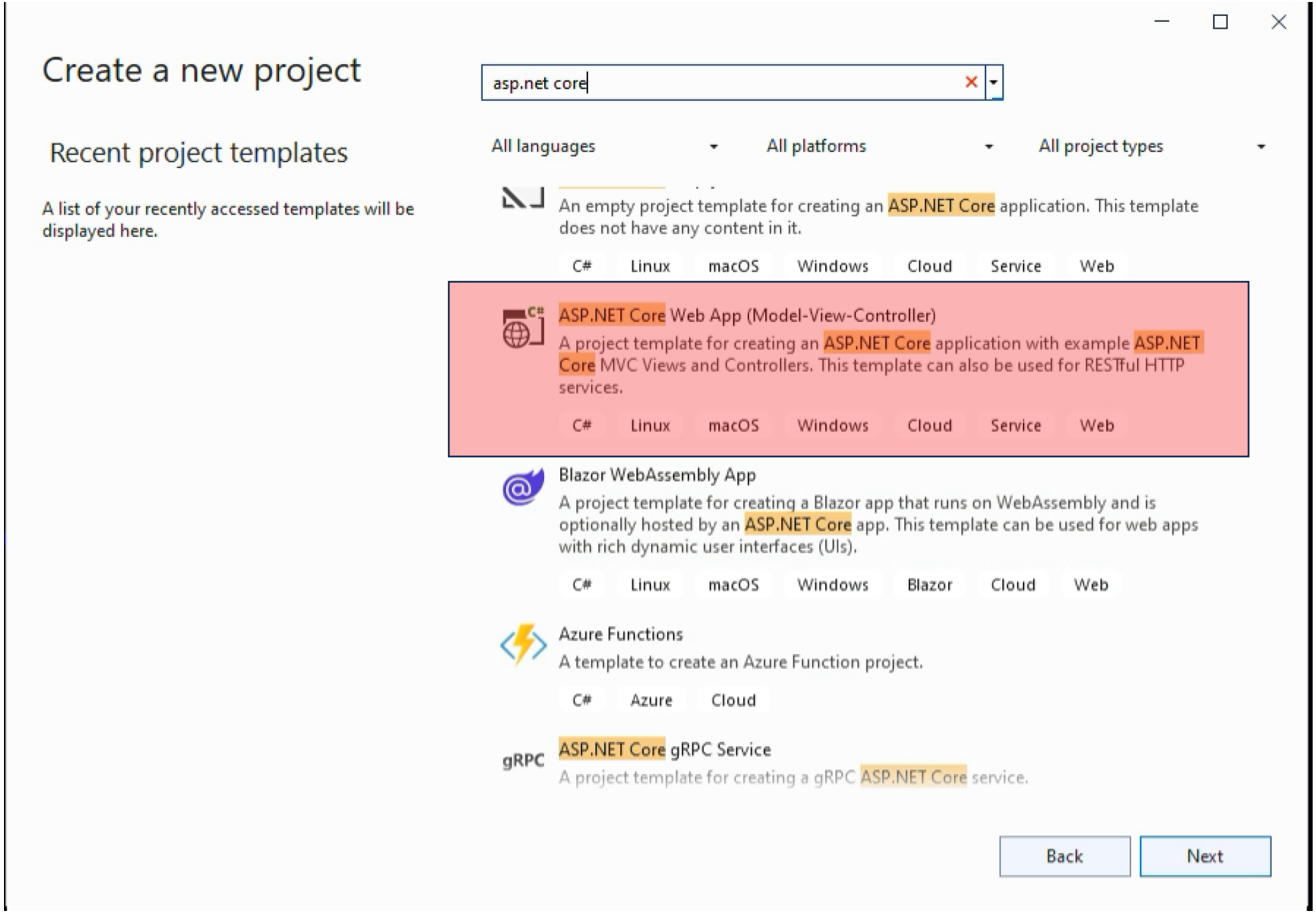
```
SELECT * FROM Students
WHERE id > 1;
```

ASP.NET Database connectivity

- ADO.NET
  - ADO.NET is a low-level data access framework that provides direct access to the database using data providers such as SqlConnection and SqlCommand for SQL Server.
  - ADO.NET provides maximum flexibility, making it suitable for scenarios where you need to execute highly optimized or complex database operations.
- Entity Framework (EF)
  - Entity Framework is a high-level Object-Relational Mapping (ORM) framework that allows developers to work with databases using .NET objects and classes, abstracting much of the database-specific code.
  - EF can work with various database systems, not just SQL Server, with providers like Entity Framework Core supporting multiple database platforms.
  - EF provides built-in change tracking, making it easier to manage updates, inserts, and deletes. It automatically generates the necessary SQL statements.
  - While EF is excellent for most common database operations, it may not offer the same level of control and optimization as handcrafted SQL for complex queries.

ASP.Net ADO.NET example

- In next couple of slides, we will go through step by step to create a web api that interacts with a student database
  - The API should be able to create table in the database
  - The API should be able to insert data in the database
  - The API should be able to delete data in the database
  - The API should be able to update date in the database
  - The API should be able to query in the database


Create a new project
asp.net core
Recent project templates
A list of your recently accessed templates will be displayed here.
All languages
All platforms
All project types
An empty project template for creating an ASP.NET Core application. This template does not have any content in it.
C# Linux macOS Windows Cloud Service Web
ASP.NET Core Web App (Model-View-Controller)
A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.
C# Linux macOS Windows Cloud Service Web
Blazor WebAssembly App
A project template for creating a Blazor app that runs on WebAssembly and is optionally hosted by an ASP.NET Core app. This template can be used for web apps with rich dynamic user interfaces (UIs).
C# Linux macOS Windows Blazor Cloud Web
Azure Functions
A template to create an Azure Function project.
C# Azure Cloud
gRPC
ASP.NET Core gRPC Service
A project template for creating a gRPC ASP.NET Core service.
Back
Next

Configure your new project

ASP.NET Core Web App (Model-View-Controller)

C# Linux macOS Windows Cloud Service Web

Project name

LocalDBWebAPI

Location

C:\Users\283602K\source\repos ...

Solution

Create new solution

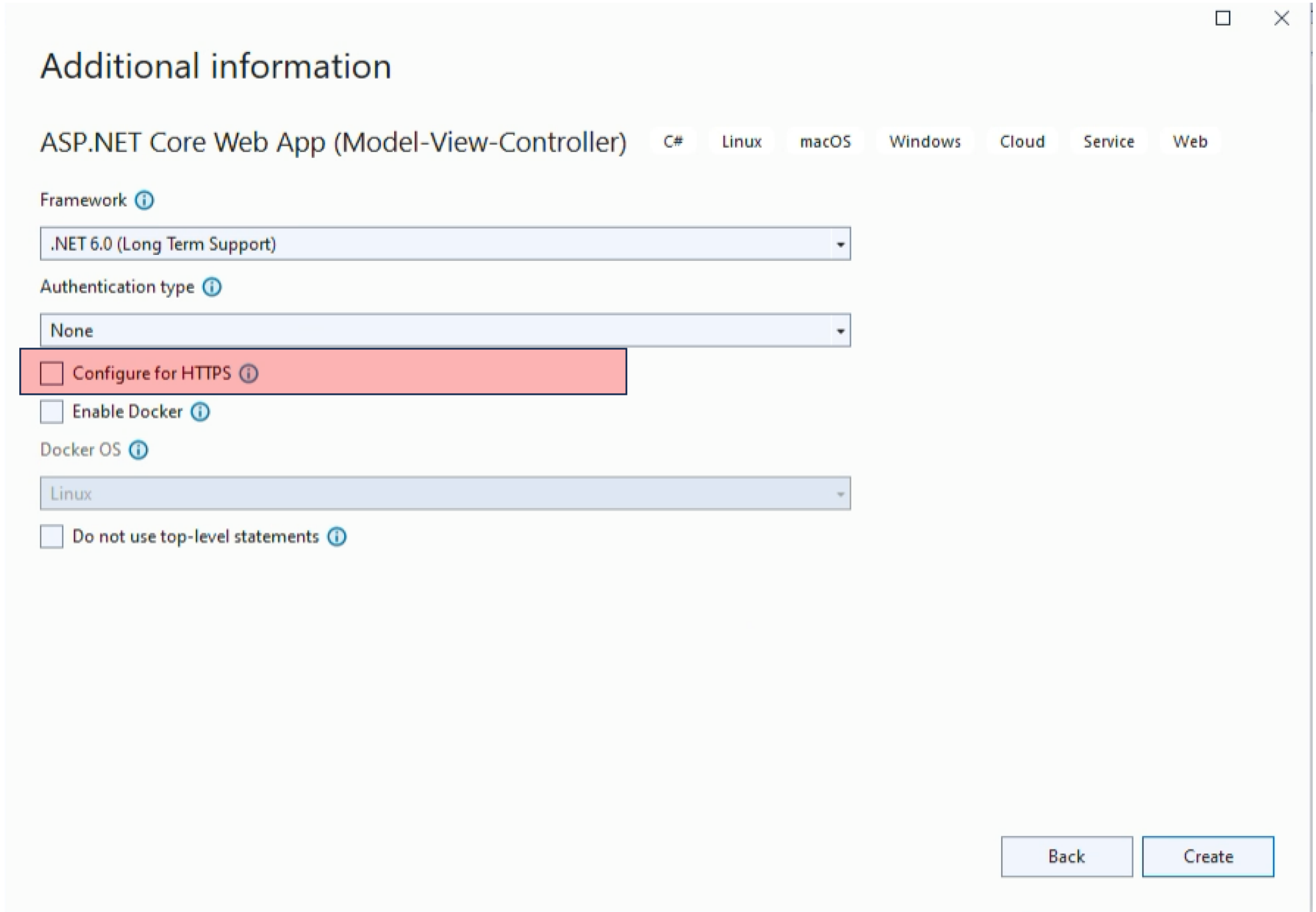
Solution name

LocalDBWeb

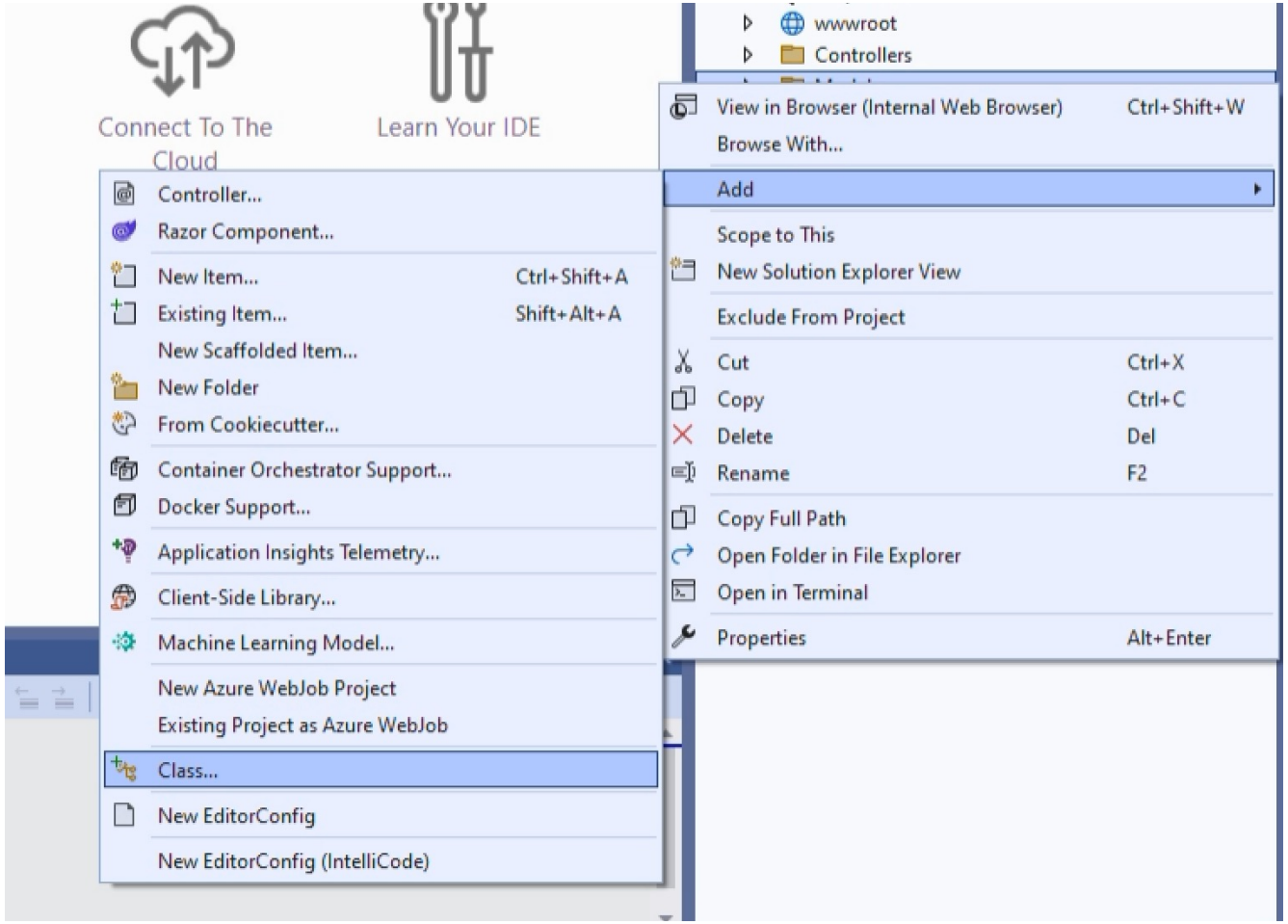
Place solution and project in the same directory

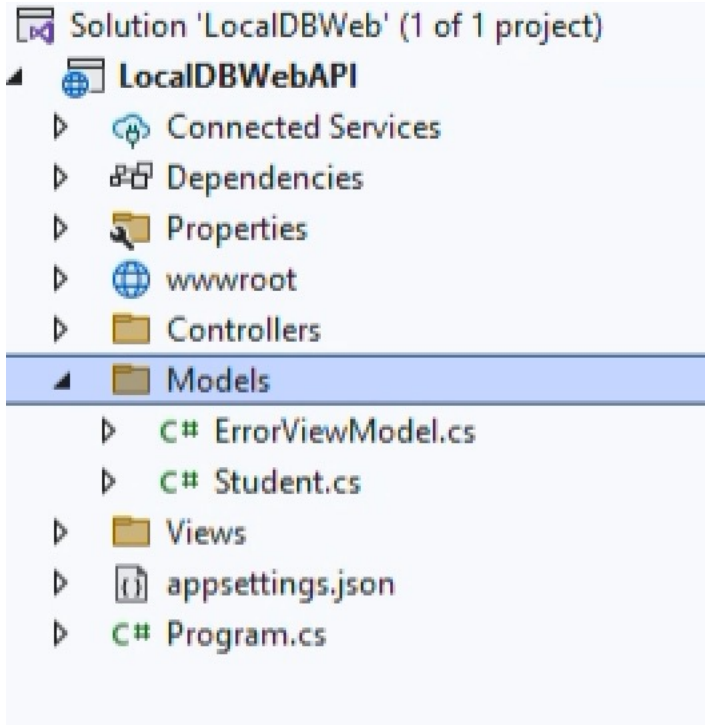
Project will be created in "C:\Users\283602K\source\repos\LocalDBWeb\LocalDBWebAPI\"

Back Next


Additional information
ASP.NET Core Web App (Model-View-Controller)
C#
Linux
macOS
Windows
Cloud
Service
Web
Framework
.NET 6.0 (Long Term Support)
Authentication type
None
Configure for HTTPS
Enable Docker
Docker OS
Linux
Do not use top-level statements
Back
Create

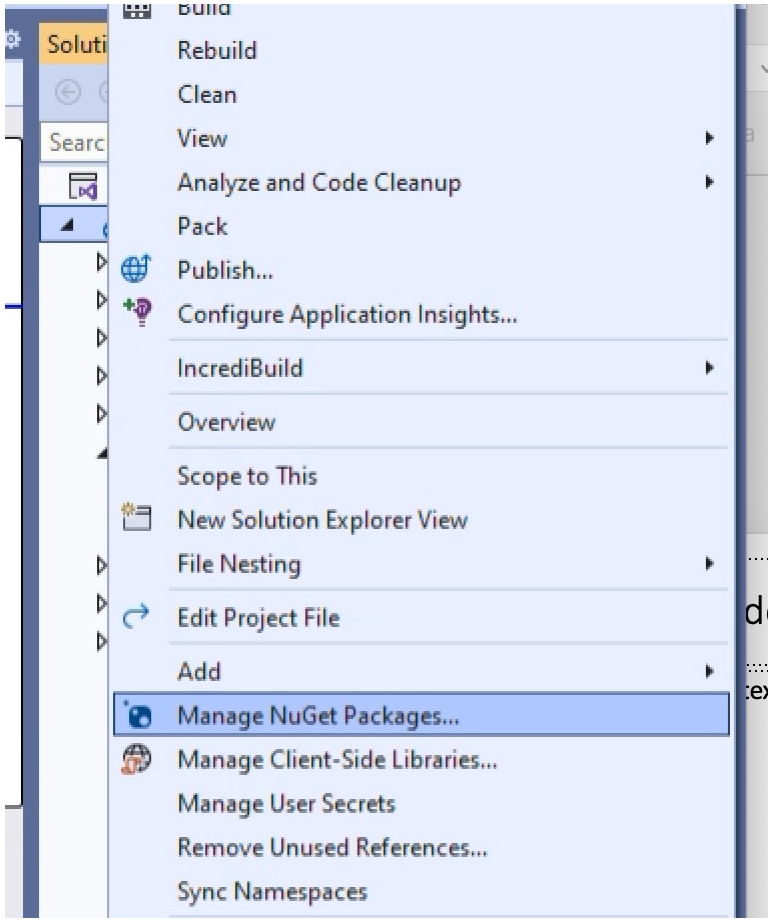
Adding Student Model class

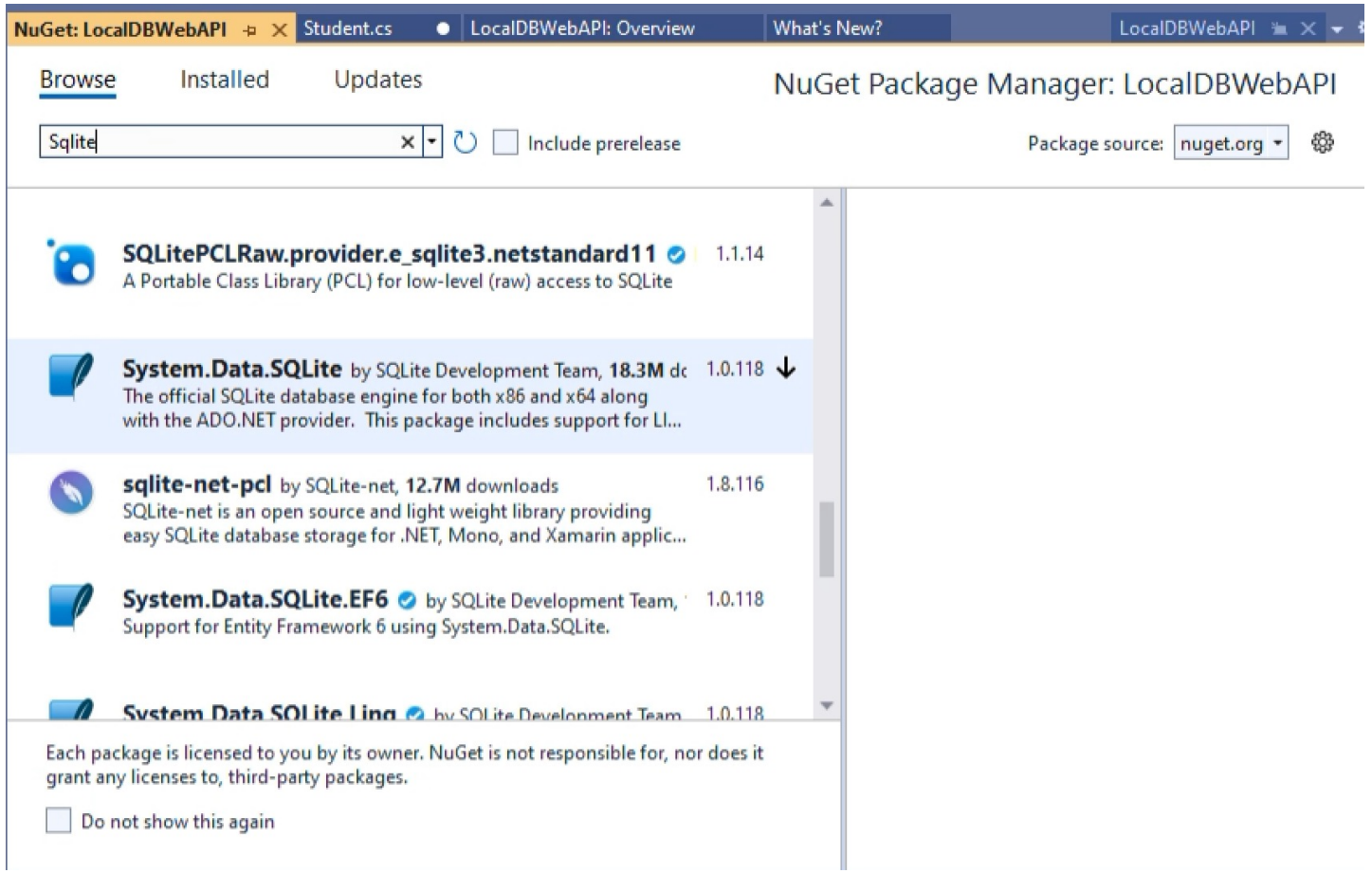





```
namespace LocalDBWebAPI.Models
{
    public class Student
    {
        public int Id { get; set; }
        public string? Name { get; set; }
        public int Age { get; set; }

    }
}
```

Adding Sqlite library

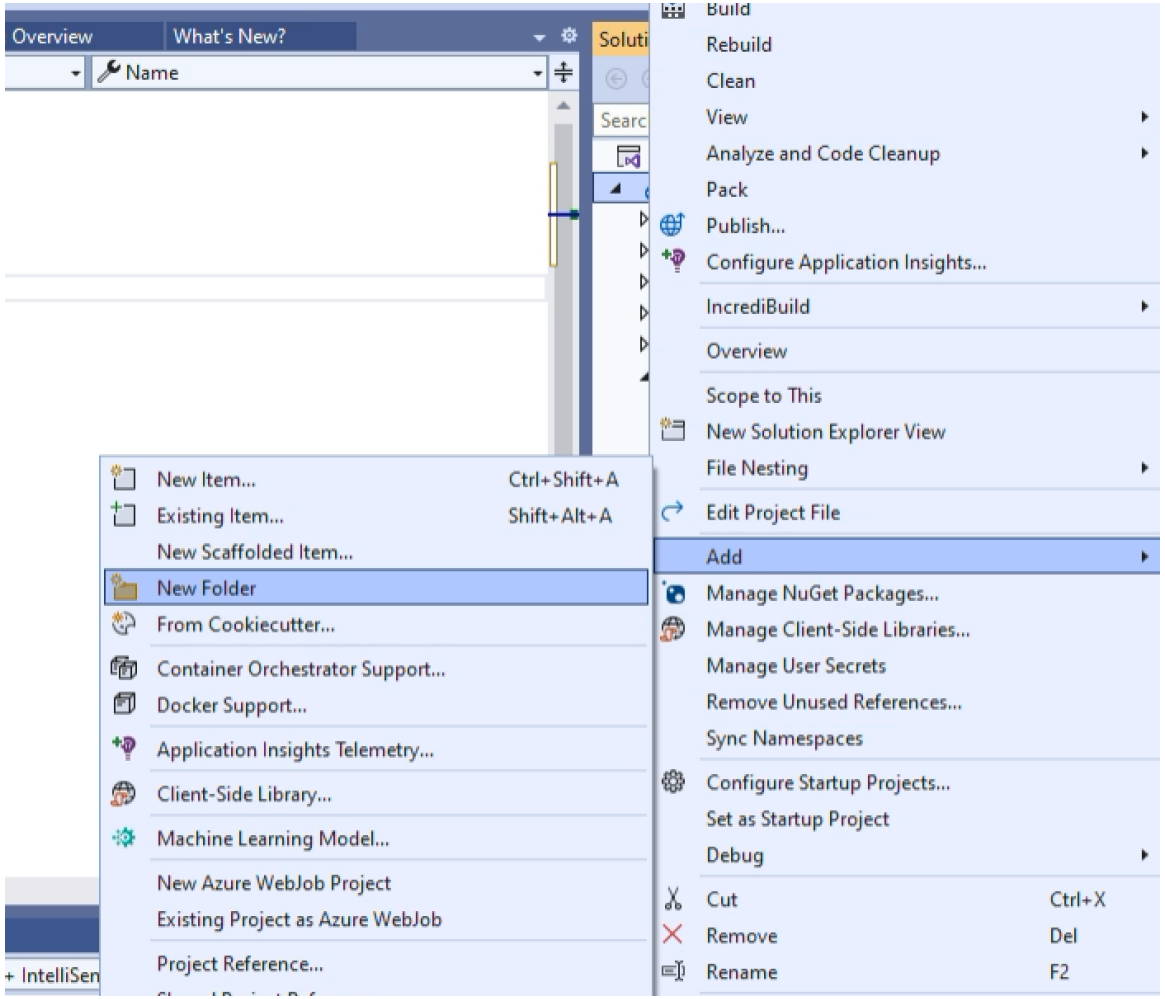


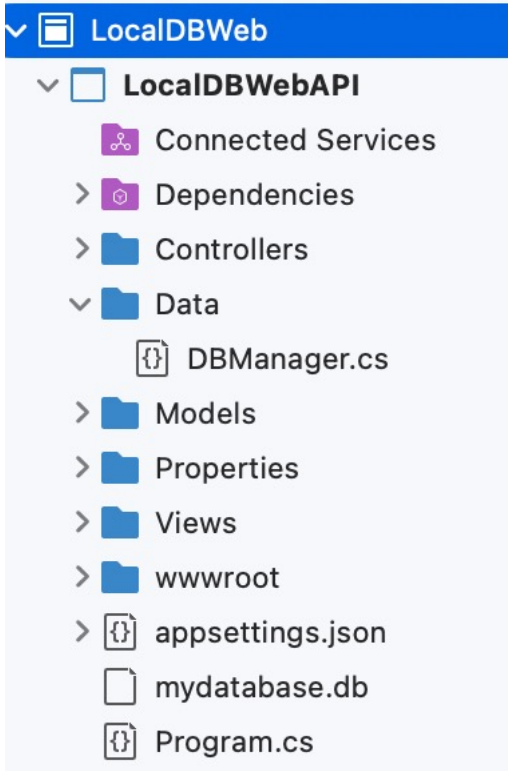


System.Data.Sqlite

- System.Data.Sqlite is cross-platform and compatible with various .NET implementations, including .NET Framework, .NET Core, and .NET 5/6. It also supports Xamarin for mobile app development.
- The library is available as a NuGet package, making it easy to add SQLite database support to your .NET projects.
- System.Data.Sqlite is implemented as an ADO.NET data provider, which means it follows the ADO.NET pattern for database access. Developers can use familiar classes like SQLiteConnection, SQLiteCommand, and SQLiteDataAdapter to interact with SQLite databases.

Creating the DBManager class(I prefer a Data folder – optinal)





SQLiteConnection

- SQLiteConnection object represents a connection to an SQLite database, and the connection string specifies various connection details, such as the database file location, version, and other parameters needed to establish the connection.

```
string connectionString = "Data Source=mydatabase.db;Version=3;";
```

```
SQLiteConnection connection = new SQLiteConnection(connectionString);
```

- In this example, the connection string specifies that the SQLite database is located in a file named mydatabase.db, and it's an SQLite version 3 database.
- After creating the SQLiteConnection object and initializing it with the connection string, you can use this connection to execute SQL commands, queries, and other database operations by creating and executing **SQLiteCommand** objects.

DBManager.cs – Create table

When you prefix a string with @, it tells the compiler to interpret the string as a verbatim string, which means that escape sequences (like \n for a newline or \" for a double quote) are treated as literal characters

ExecuteNonQuery method is used to execute SQL commands that don't return data

```
namespace LocalDBWebAPI.Data
{
    public class DBManager
    {
        private static string connectionString = "Data Source=mydatabase.db;Version=3;";

        public static bool CreateTable()
        {
            try
            {
                using (SQLiteConnection connection = new SQLiteConnection(connectionString))
                {
                    connection.Open();
                    // Create a new SQLite command to execute SQL
                    using (SQLiteCommand command = connection.CreateCommand())
                    {
                        // SQL command to create a table named "StudentTable"
                        command.CommandText = @"
                    CREATE TABLE StudentTable (
                        ID INTEGER,
                        Name TEXT,
                        Age INTEGER
                    )";
                        // Execute the SQL command to create the table
                        command.ExecuteNonQuery();
                        connection.Close();
                    }
                }
                Console.WriteLine("Table created successfully.");
                return true;
            }
            catch (Exception ex)
            {
                Console.WriteLine("Error: " + ex.Message);
            }
            return false; // Create table failed
        }
```

DBManager.cs – Insert

The AddWithValue method of the Parameters collection is a convenient way to add parameters and their values simultaneously. This method helps protect against SQL injection and ensures that values are properly parameterized.

@ symbol in front of the parameter names indicates that they are placeholders.

```
public static bool Insert(Student student)
{
    try
    {
        // Create a new SQLite connection
        using (SQLiteConnection connection = new SQLiteConnection(connectionString))
        {
            connection.Open();

            // Create a new SQLite command to execute SQL
            using (SQLiteCommand command = connection.CreateCommand())
            {
                // SQL command to insert data into the "StudentTable" table
                command.CommandText = @"
                INSERT INTO StudentTable (ID, Name, Age)
                VALUES (@ID, @Name, @Age)";

                // Define parameters for the query
                command.Parameters.AddWithValue("@ID", student.Id);
                command.Parameters.AddWithValue("@Name", student.Name);
                command.Parameters.AddWithValue("@Age", student.Age);

                // Execute the SQL command to insert data
                int rowsInserted = command.ExecuteNonQuery();

                // Check if any rows were inserted
                connection.Close();
                if (rowsInserted > 0)
                {
                    return true; // Insertion was successful
                }
            }
            connection.Close();
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine("Error: " + ex.Message);
    }

    return false; // Insertion failed
}
```

DBManager.cs – Delete

```
public static bool Delete(int id)
{
    try
    {
        // Create a new SQLite connection
        using (SQLiteConnection connection = new SQLiteConnection(connectionString))
        {
            connection.Open();

            // Create a new SQLite command to execute SQL
            using (SQLiteCommand command = connection.CreateCommand())
            {
                // Build the SQL command to delete data by ID
                command.CommandText = $"DELETE FROM StudentTable WHERE ID = @ID";
                command.Parameters.AddWithValue("@ID", id);

                // Execute the SQL command to delete data
                int rowsDeleted = command.ExecuteNonQuery();

                // Check if any rows were deleted
                connection.Close();
                if (rowsDeleted > 0)
                {
                    return true; // Deletion was successful
                }
            }
            connection.Close();
        }

        return false; // No rows were deleted
    }
    catch (Exception ex)
    {
        Console.WriteLine("Error: " + ex.Message);
        return false; // Deletion failed
    }
}
```

DBManager.cs – Update

```
public static bool Update(Student student)
{
    try
    {
        // Create a new SQLite connection
        using (SQLiteConnection connection = new SQLiteConnection(connectionString))
        {
            connection.Open();

            // Create a new SQLite command to execute SQL
            using (SQLiteCommand command = connection.CreateCommand())
            {
                // Build the SQL command to update data by ID
                command.CommandText = $"UPDATE StudentTable SET Name = @Name, Age = @Age WHERE ID = @ID";
                command.Parameters.AddWithValue("@Name", student.Name);
                command.Parameters.AddWithValue("@Age", student.Age);
                command.Parameters.AddWithValue("@ID", student.Id);

                // Execute the SQL command to update data
                int rowsUpdated = command.ExecuteNonQuery();
                connection.Close();
                // Check if any rows were updated
                if (rowsUpdated > 0)
                {
                    return true; // Update was successful
                }
            }
            connection.Close();
        }

        return false; // No rows were updated
    }
    catch (Exception ex)
    {
        Console.WriteLine("Error: " + ex.Message);
        return false; // Update failed
    }
}
```

DBManager.cs – Select all

you can use the same ExecuteReader method to execute SQL queries and retrieve a SQLiteDataReader.

SQLiteDataReader.Read() method is used to advance the data reader to the next row in the result set. It returns true if there are more rows to read, and false if there are no more rows (i.e., you've reached the end of the result set).

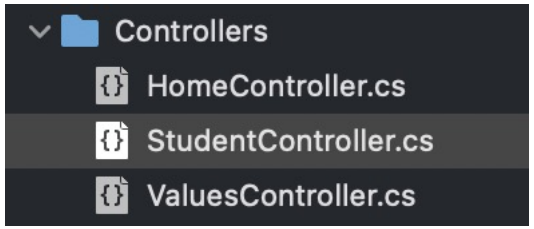
```
public static List<Student> GetAll()
{
    List<Student> studentList = new List<Student>();

    try
    {
        // Create a new SQLite connection
        using (SQLiteConnection connection = new SQLiteConnection(connectionString))
        {
            connection.Open();

            // Create a new SQLite command to execute SQL
            using (SQLiteCommand command = connection.CreateCommand())
            {
                // Build the SQL command to select all student data
                command.CommandText = "SELECT * FROM StudentTable";

                // Execute the SQL command and retrieve data
                using (SQLiteDataReader reader = command.ExecuteReader())
                {
                    while (reader.Read())
                    {
                        Student student = new Student();
                        student.Id = Convert.ToInt32(reader["ID"]);
                        student.Name = reader["Name"].ToString();
                        student.Age = Convert.ToInt32(reader["Age"]);

                        // Create a Student object and add it to the list
                        studentList.Add(student);
                    }
                }
            }
            connection.Close();
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine("Error: " + ex.Message);
    }

    return studentList;
}
```

DBManager.cs – Select by id

```
public static Student GetById(int id)
{
    Student student = null;

    try
    {
        // Create a new SQLite connection
        using (SQLiteConnection connection = new SQLiteConnection(connectionString))
        {
            connection.Open();

            // Create a new SQLite command to execute SQL
            using (SQLiteCommand command = connection.CreateCommand())
            {
                // Build the SQL command to select a student by ID
                command.CommandText = "SELECT * FROM StudentTable WHERE ID = @ID";
                command.Parameters.AddWithValue("@ID", id);

                // Execute the SQL command and retrieve data
                using (SQLiteDataReader reader = command.ExecuteReader())
                {
                    if (reader.Read())
                    {
                        student = new Student();
                        student.Id = Convert.ToInt32(reader["ID"]);
                        student.Name = reader["Name"].ToString();
                        student.Age = Convert.ToInt32(reader["Age"]);

                    }
                }
            }
            connection.Close();
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine("Error: " + ex.Message);
    }

    return student;
}
```

Now add a controller (Web API)

- Controller -> Add controller -> API -> API Controller Empty



API Endpoint:

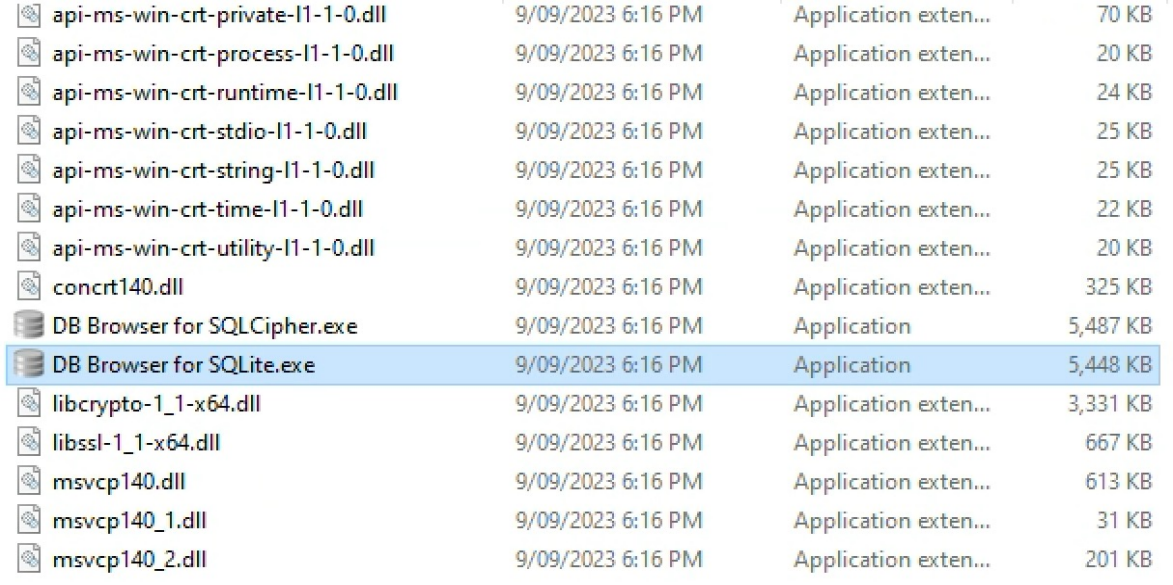
http://localhost:port/api/student/

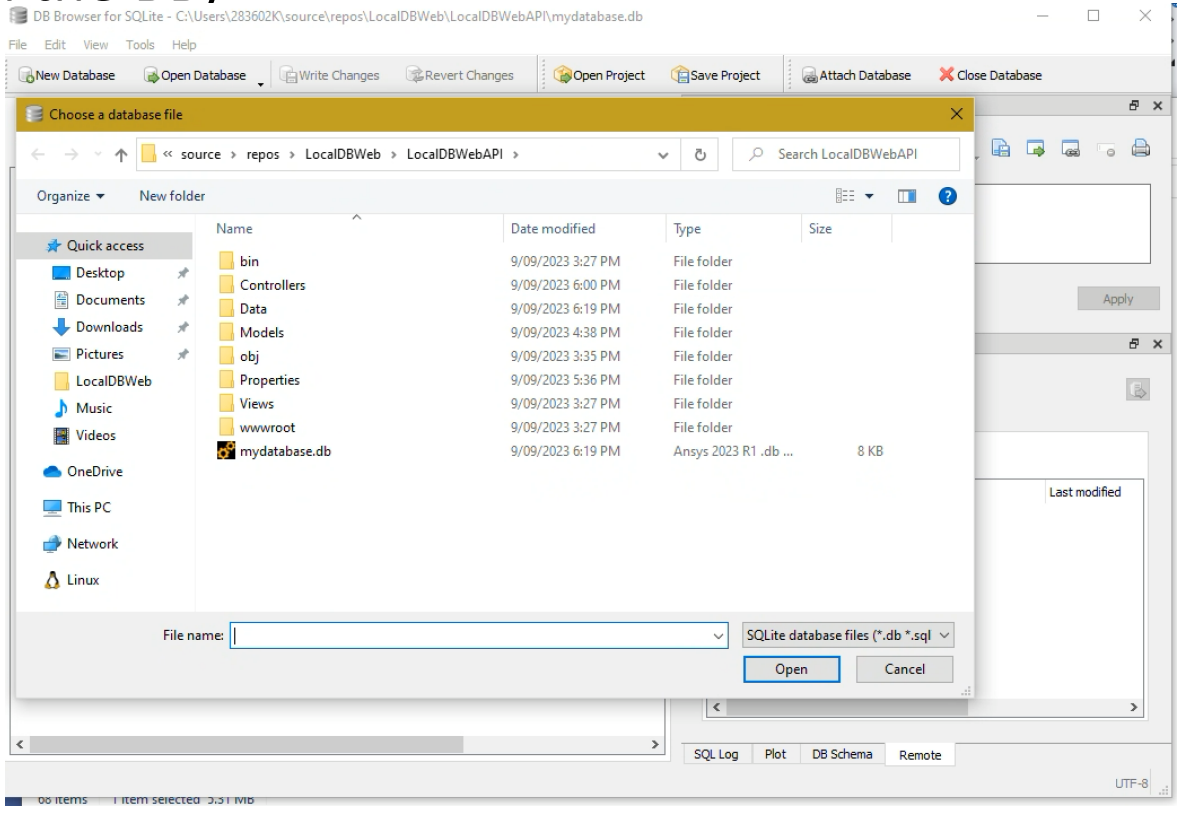
```
[HttpDelete]
[Route("{id}")]
public IActionResult Delete(int id)
{
    if(DBManager.Delete(id))
    {
        return Ok("Successfully Deleted");
    }
    return BadRequest("Could not delete");
}
[HttpPut]
public IActionResult Update(Student student)
{
    if (DBManager.Update(student))
    {
        return Ok("Successfully updated");
    }
    return BadRequest("Could not update");
}
```

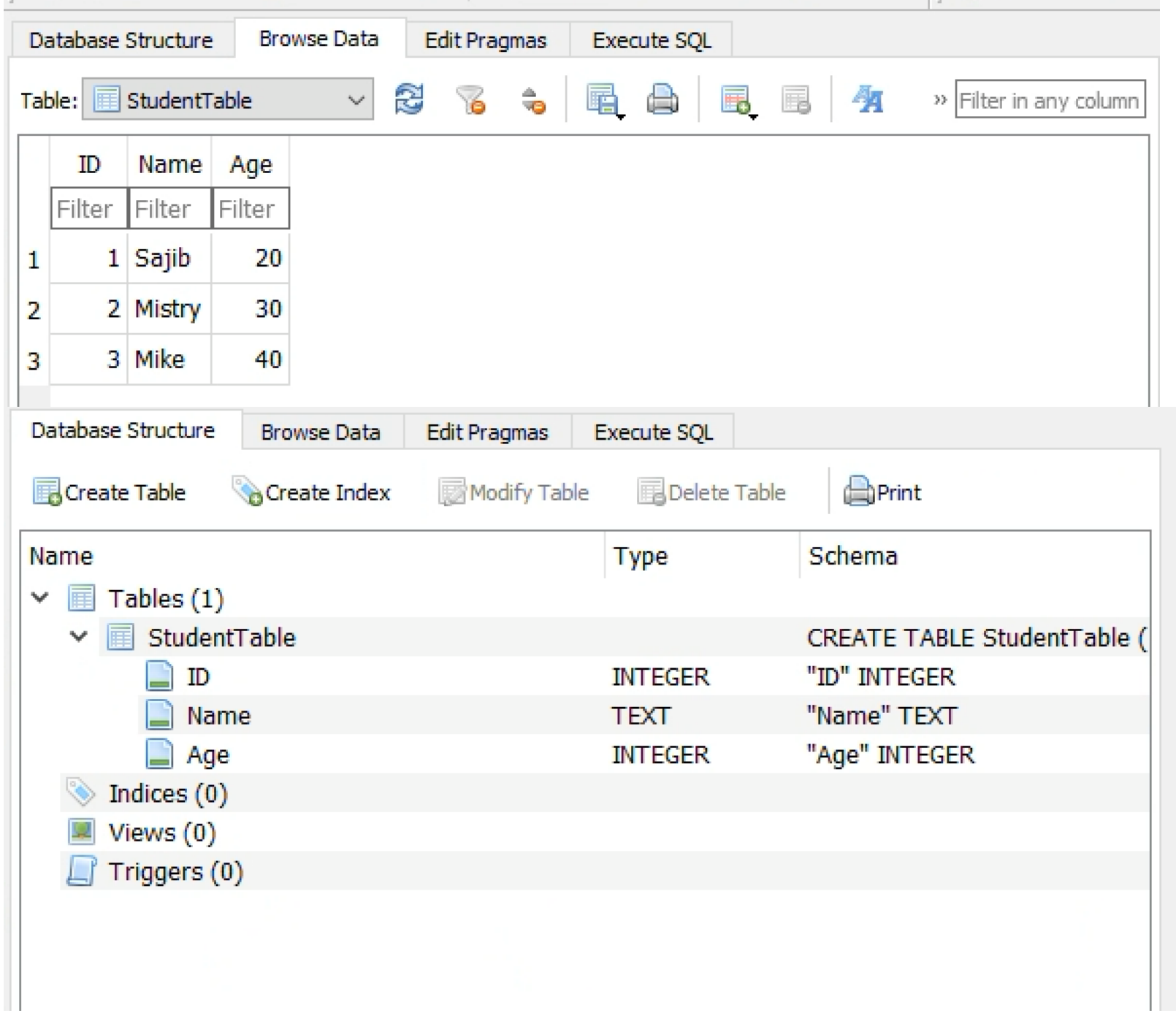
```
using LocalDBWebAPI.Data;
using LocalDBWebAPI.Models;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;

namespace LocalDBWebAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class StudentController : ControllerBase
    {
        [HttpGet]
        public IEnumerable<Student> getStudents()
        {
            List<Student> students = DBManager.GetAll();
            return students;
        }
        [HttpGet]
        [Route("{id}")]
        public IActionResult Get(int id)
        {
            Student student = DBManager.GetById(id);
            if(student == null)
            {
                return NotFound();
            }
            return Ok(student);
        }
        [HttpPost]
        public IActionResult Post([FromBody] Student student)
        {
            if (DBManager.Insert(student))
            {
                return Ok("Successfully inserted");
            }
            return BadRequest("Error in data insertion");
        }
```

How do you check that it is working?

- Use POSTMAN (Check lecture 6)
- Or explore the created db (i.e., mydatabase.db)
  - DB Browser for SQLite (DB4S) is a high quality, visual, open source tool to create, design, and edit database files compatible with SQLite.
  - https://sqlitebrowser.org (zip attached in the BB)

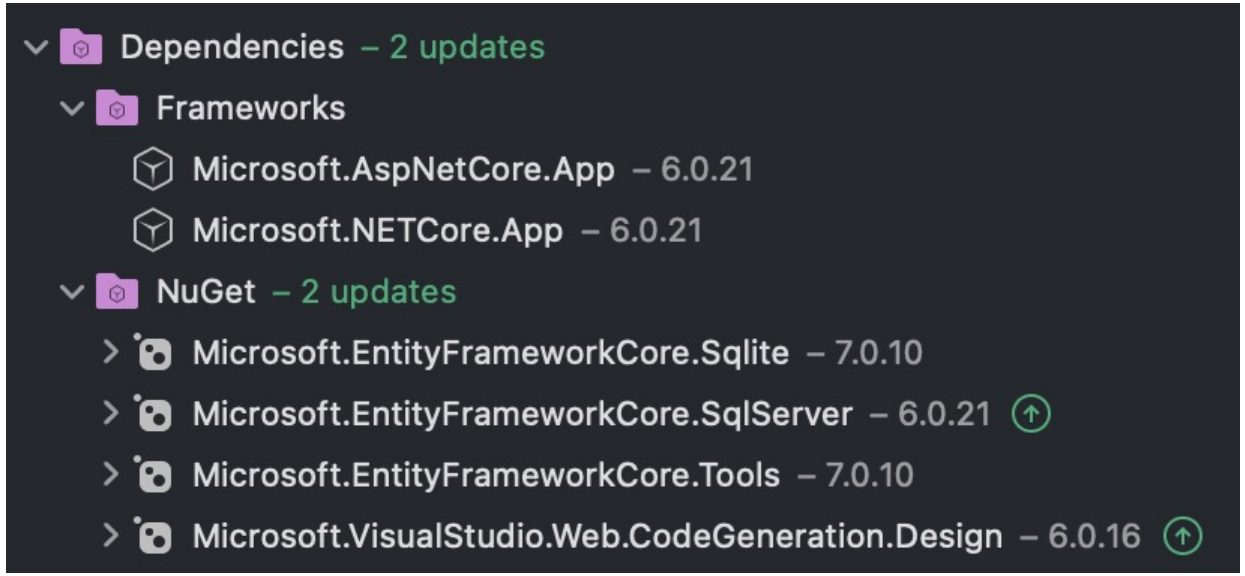





Database Structure
Browse Data
Edit Pragmas
Execute SQL
Table: StudentTable
Filter in any column
ID
Name
Age
Filter
Filter
Filter
1 1 Sajib 20
2 2 Mistry 30
3 3 Mike 40
Database Structure
Browse Data
Edit Pragmas
Execute SQL
Create Table
Create Index
Modify Table
Delete Table
Print
Name
Type
Schema
Tables (1)
StudentTable
CREATE TABLE StudentTable (
ID
INTEGER
"ID" INTEGER
Name
TEXT
"Name" TEXT
Age
INTEGER
"Age" INTEGER
Indices (0)
Views (0)
Triggers (0)

Entity Framework (EF) – example
Simplified and no coding at all !!

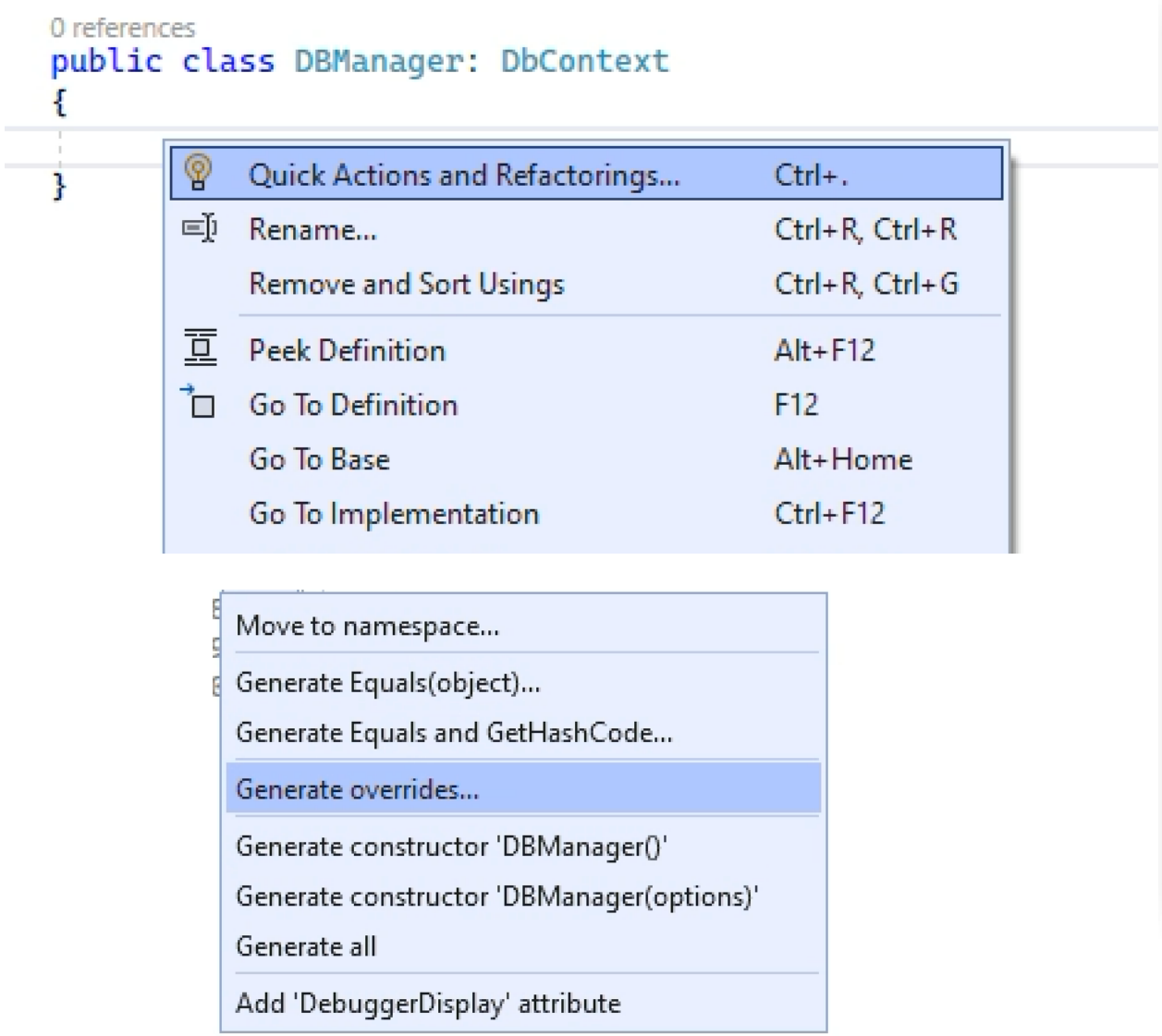
- Create a different ASP.NET core web (Model-view-controller) project (LocalDBWebApiUsingEF in the example source code)
- Add the student model
- Add four NuGet packages – start with Sqlite and Tools. Other two should be available (if not, install or update according to the project's .Net version 6.0 or 7.0)

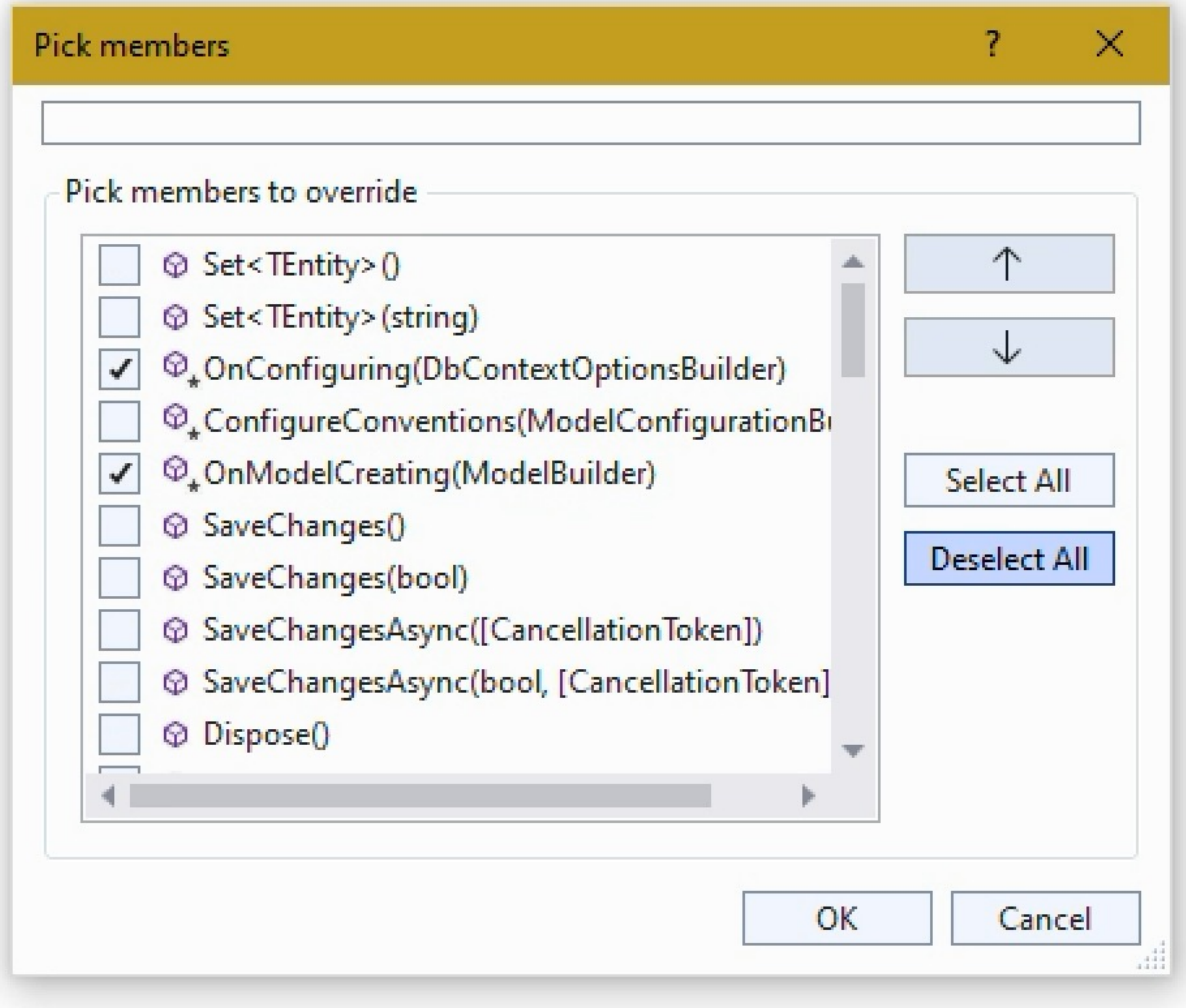


Create DB Manager

- Create a Folder “Data” and add a DBManage.cs class
- Inherit from DBContext.
- DbContext is a crucial class that is part of the Entity Framework Core (EF Core) and Entity Framework 6 (EF6).
- It represents a session with the database and provides a high-level API for interacting with your database, including querying, inserting, updating, and deleting data.
- DbContext acts as a bridge between your domain classes (entities) and the database.

DBManager.cs in EF – generating overrides





DBManager.cs

Configuration: You can configure the DbContext through its constructor or by using the OnConfiguring method to specify the database provider and connection string.

In Entity Framework, the OnModelCreating method is a crucial part of the DbContext class, and it's used for configuring the entity classes and defining the database model.

Additionally, you can use this method to seed initial data into your database when the database is created or updated.

```
using LocalDBWebApiUsingEF.Models;
using Microsoft.EntityFrameworkCore;

namespace LocalDBWebApiUsingEF.Data
{
    public class DBManager : DbContext
    {
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlite(@"Data Source = Students.db;");
        }

        public DbSet<Student> Students { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            List<Student> students = new List<Student>();
            Student student = new Student();
            student.Id = 1;
            student.Name = "Sajib";
            student.Age = 20;
            students.Add(student);

            student = new Student();
            student.Id = 2;
            student.Name = "Mistry";
            student.Age = 30;
            students.Add(student);

            student = new Student();
            student.Id = 3;
            student.Name = "Mike";
            student.Age = 40;
            students.Add(student);
            modelBuilder.Entity<Student>().HasData(students);
        }
    }
}
```

Update Program.cs

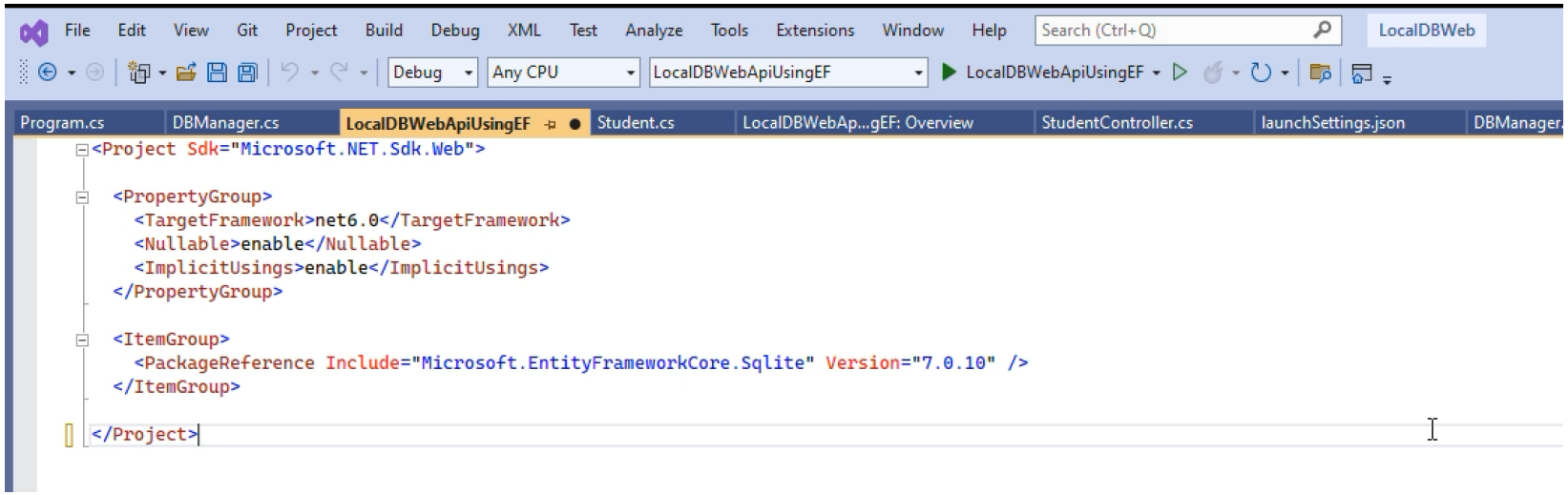
- Tell the server about your DB.

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();
builder.Services.AddDbContext<DBManager>();
```

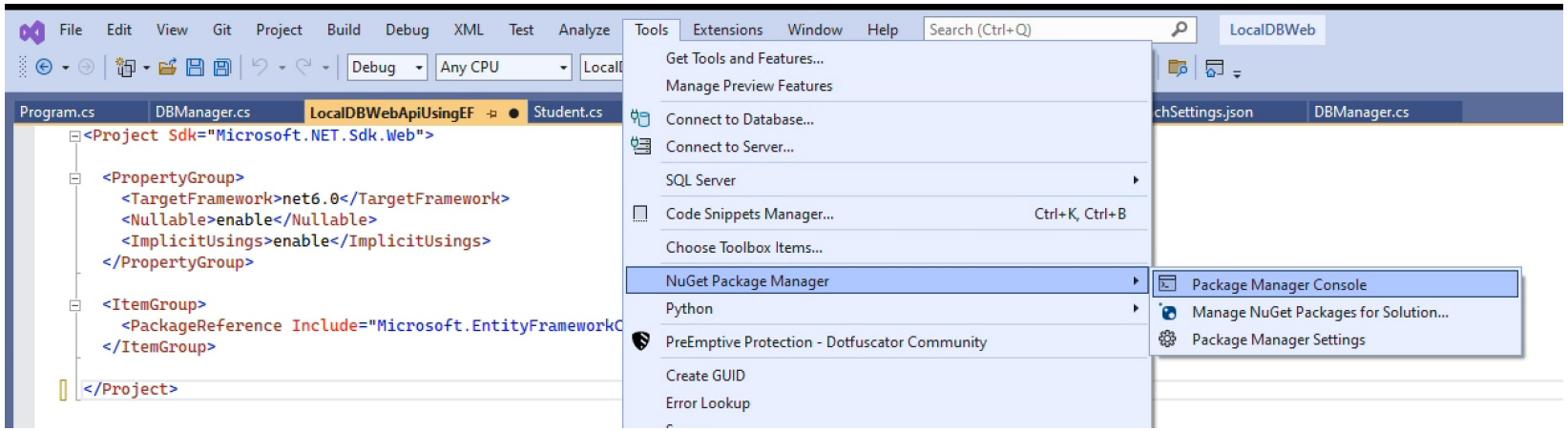
- By using builder.Services.AddDbContext, you're registering your DbContext with the ASP.NET Core dependency injection container, which allows you to inject it into other parts of your application and ensures that it's properly managed in terms of its lifetime and disposal.

Database Migration

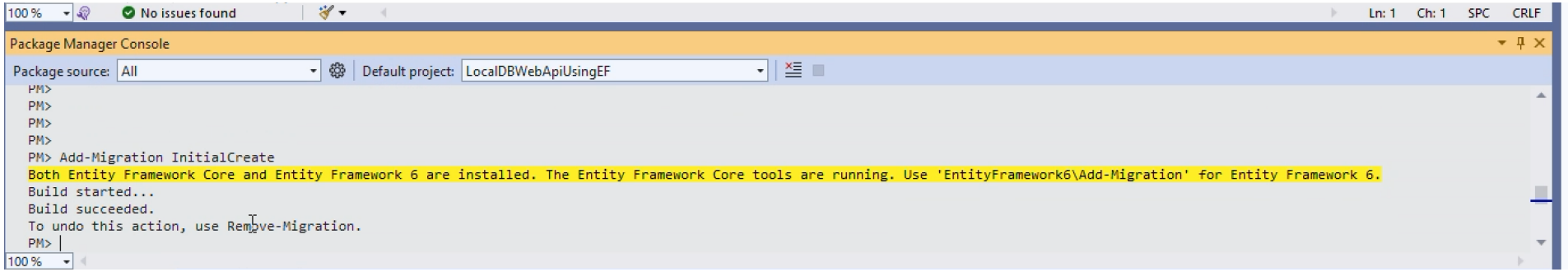
- DbContext can be used in conjunction with migrations to create and update the database schema based on changes in your entity classes.
- Entity framework tools to rescue.
- Make sure you are building the EF project.



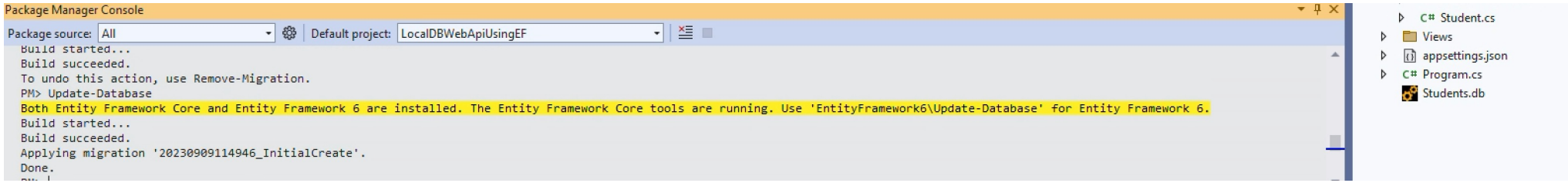
1. Tools->NuGetPackage Manager -> Package Manager Console



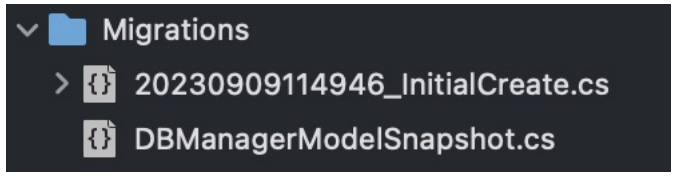
2. Command: <Add-Migration InitialCreate>



3. Command: <update-Database>



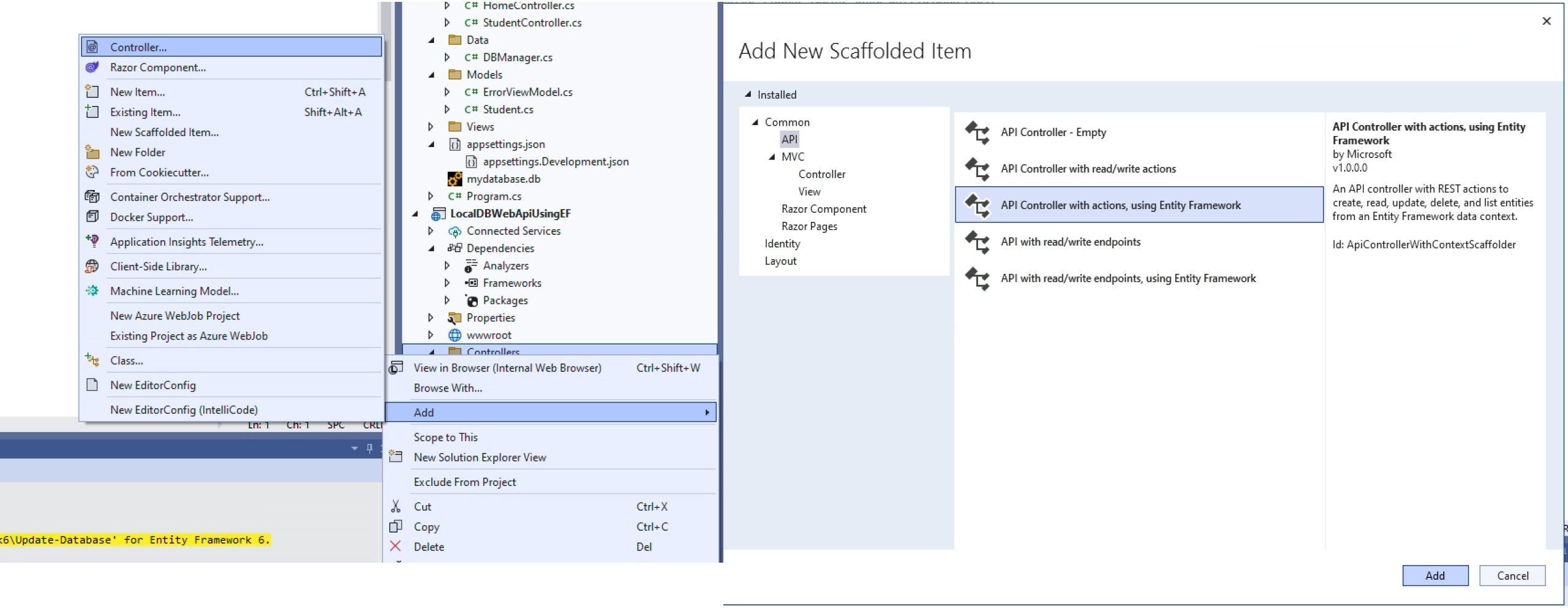
Migration folder and db are created now



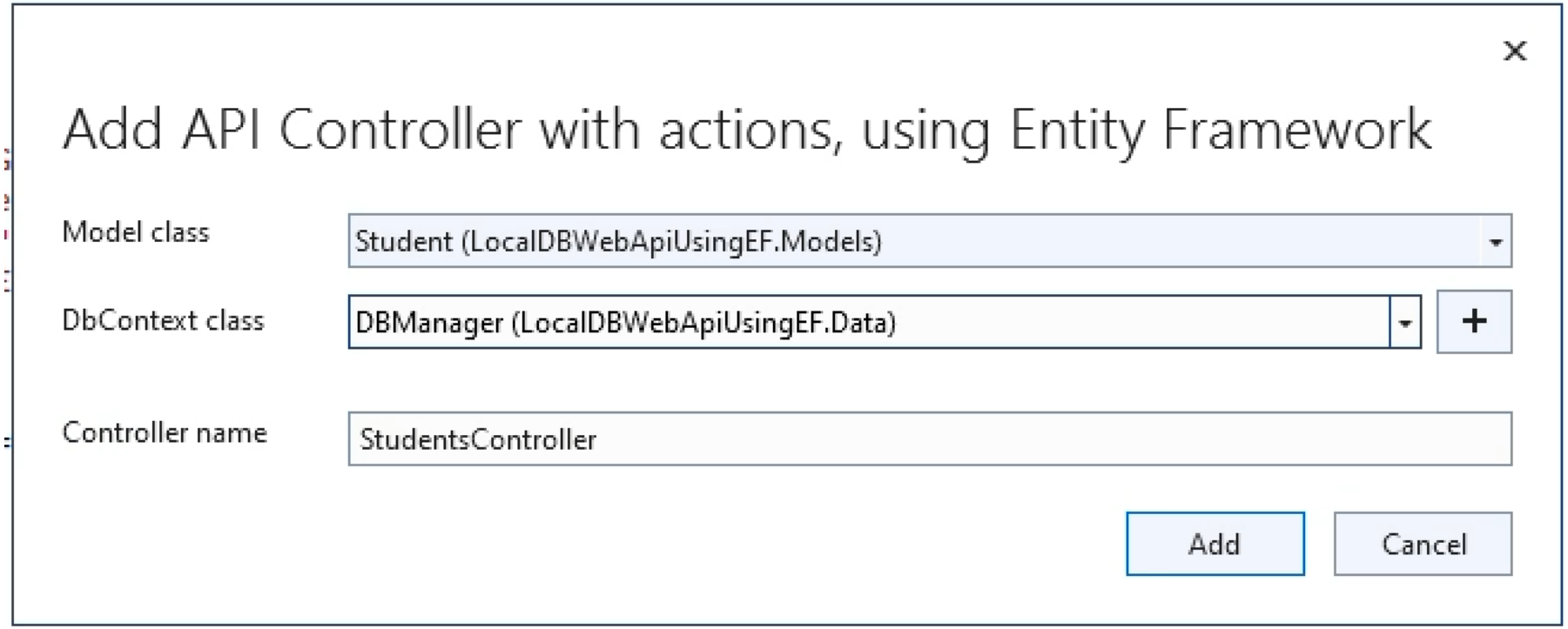
Autogenerated!!

```
namespace LocalDBWebApiUsingEF.Migrations
{
    /// <inheritdoc />
    public partial class InitialCreate : Migration
    {
        /// <inheritdoc />
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Students",
                columns: table => new
                {
                    Id = table.Column<int>(type: "INTEGER", nullable: false)
                        .Annotation("Sqlite:Autoincrement", true),
                    Name = table.Column<string>(type: "TEXT", nullable: true),
                    Age = table.Column<int>(type: "INTEGER", nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Students", x => x.Id);
                });

            migrationBuilder.InsertData(
                table: "Students",
                columns: new[] { "Id", "Age", "Name" },
                values: new object[,]
                {
                    { 1, 20, "Sajib" },
                    { 2, 30, "Mistry" },
                    { 3, 40, "Mike" }
                });
        }

        /// <inheritdoc />
        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropTable(
                name: "Students");
        }
    }
}
```

Create Controller (API controller with actions, using Entity Framework)



Scaffolding – code will be autogenerated



- If required, add [FromBody] attribute for Post !!

StudentsController.cs

```cs
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using LocalDBWebApiUsingEF.Data;
using LocalDBWebApiUsingEF.Models;

namespace LocalDBWebApiUsingEF.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class StudentsController : ControllerBase
    {
        private readonly DBManager _context;

        public StudentsController(DBManager context)
        {
            _context = context;
        }

        // GET: api/Students
        [HttpGet]
        public async Task<ActionResult<IEnumerable<Student>>> GetStudents()
        {
          if (_context.Students == null)
          {
              return NotFound();
          }
            return await _context.Students.ToListAsync();
        }

        // GET: api/Students/5
        [HttpGet("{id}")]
        public async Task<ActionResult<Student>> GetStudent(int id)
        {
          if (_context.Students == null)
          {
              return NotFound();
          }
            var student = await _context.Students.FindAsync(id);

            if (student == null)
            {
                return NotFound();
            }

            return student;
        }
```

```cs
[HttpPost]
public async Task<ActionResult<Student>> PostStudent(Student student)
{
  if (_context.Students == null)
  {
      return Problem("Entity set 'DBManager.Students'  is null.");
  }
    _context.Students.Add(student);
    await _context.SaveChangesAsync();

    return CreatedAtAction("GetStudent", new { id = student.Id }, student);
}

// DELETE: api/Students/5
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteStudent(int id)
{
    if (_context.Students == null)
    {
        return NotFound();
    }
    var student = await _context.Students.FindAsync(id);
    if (student == null)
    {
        return NotFound();
    }

    _context.Students.Remove(student);
    await _context.SaveChangesAsync();

    return NoContent();
}

private bool StudentExists(int id)
{
    return (_context.Students?.Any(e => e.Id == id)).GetValueOrDefault();
}
```

Test using Postman and DB Browser

- Hope you have enjoyed this session.