CRUD API using ASP.NET Core MVC Web APIs with SQL Server and POSTMAN

Step 1: Install All Required Software

a. .NET SDK and Visual Studio

- **Purpose**: To create, build, and run ASP.NET Core Web API projects.
- **Download Link**:
  - .NET SDK + Visual Studio Community
- **During installation, select the following workloads**:
  - ASP.NET and web development ✅
  - .NET desktop development ✅
  - Optional: Data storage and processing ✅ (if planning to use SQL Server tools from inside VS)

b. Install SQL Server + SSMS (SQL Server Management Studio)

- **SQL Server 2022 Express** (Free version)
  - **Purpose**: To host your SQL Server database locally.
  - **Download Link**: https://www.microsoft.com/en-us/sql-server/sql-server-downloads
- **SSMS (SQL Server Management Studio)**
  - **Purpose**: GUI to create, view, and manage your database, tables, and data.
  - **Download Link**: SSMS Download Page
  - If the above doesn't work, try: https://aka.ms/ssmsfullsetup

c. Install Postman

- **Purpose**: To test all your HTTP API endpoints (GET, POST, PUT, DELETE).
- **Download Link**: https://www.postman.com/downloads/

Step 2: Create a New ASP.NET Core Web API Project

1. Open **Visual Studio**.
2. Click **Create a new project**.
3. Search for **ASP.NET Core Web API**, then click **Next**.
4. Configure the project:
   - **Project Name**: `CrudApiDemo`
   - **Location**: `LocalDrive/SAGAR/Practice`
   - Click **Next**.
5. Configure project settings:
   - **Framework**: Choose `.NET 6` or `.NET 7`
   - **Uncheck**: Enable OpenAPI Support (optional)
   - **Uncheck**: Enable HTTPS (optional for local testing)
   - **Check**: Use controllers (NOT minimal APIs)
   - Click **Create**

Step 3: Create SQL Server Database (Using SSMS)

a. Open SSMS and Connect to Your Local Server

1. Open Command Prompt (`cmd`) and type:
2. `sqlcmd -L`
   - This shows available SQL Server instances.
   - You'll see something like: `DESKTOP-H66TGF7\SQLEXPRESS`
3. Open SSMS.
4. In the **Connect to Server** window:
   - **Server type**: Database Engine
   - **Server name**: `DESKTOP-H65TGF7\SQLEXPRESS` (example)
   - **Authentication**: Windows Authentication
   - Click **Connect**

b. Create New Database

1. In Object Explorer, right-click `Databases` > `New Database`.
2. Name: `PracticeDb`
3. Click OK.

c. Create Employees Table

1. Expand `PracticeDb` > Right-click `Tables` > `New Query`
2. Run the following SQL:

```
CREATE TABLE Employees (
    Id INT PRIMARY KEY IDENTITY,
    Name NVARCHAR(100),
    Department NVARCHAR(50),
    Salary INT
);
```

3. Click **Execute**.

Step 4: Connect ASP.NET Core Web API to SQL Server using ADO.NET

a. Add Connection String in `appsettings.json`

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",

  "ConnectionStrings": {
    "DefaultConnection": "Server=DESKTOP-
H66TGF7\\SQLEXPRESS;Database=PracticeDB;Trusted_Connection=True;TrustServerCertificate=True;"
  }

}
```

Make sure to use double backslashes (`\\`) in the connection string for escape characters in JSON.

✅ You've now set up your SQL Server and connected it to your Web API project.

✅ Step 5: Create a `Models` Folder and Define a Model Class

✨ What is a Model?

In ASP.NET Core, a **Model** is a C# class that represents the **structure of data** in your application. It defines the properties of the data that will be stored in or retrieved from the database.

Why do we need a Model?

- It provides a clean way to represent database tables.
- Helps in data transfer between layers (UI, Database, API).
- Ensures strong typing (compile-time checking).
- Easier to manage data validation and business logic.

🛠️ How to Create a Model:

1. Right-click on your project folder > `Add` > `New Folder` > Name it: `Models`
2. Right-click on the `Models` folder > `Add` > `Class` > Name it: `Employee.cs`

👇 Example Model Class:

```
// Namespace refers to the folder structure
namespace CrudApiDemo.Models
{
    // This class represents the Employees table
    public class Employee
    {
        public int Id { get; set; }        // Primary key (auto-incremented)
        public string Name { get; set; }   // Employee name
        public string Department { get; set; } // Employee's department
        public int Salary { get; set; }    // Employee's salary
    }
}
```

✅ Step 6: Create a `Data` Folder and Write the `EmployeeDataAccess` Class

Note : Creating a Data Access Layer (DAL) is a core part of working with ADO.NET.

🔍 What is the Data Access Layer (DAL)?

The **Data Access Layer** is a **separate class or component** in your application responsible for handling **all interactions with the database**. It acts as a bridge between:

```
Controller/Business Logic  ⟷  Data Access Layer  ⟷  SQL Server
```

🔗 What is ADO.NET?

ADO.NET (ActiveX Data Objects for .NET) is a **Microsoft data access technology** used in .NET to work with databases. (**ADO.NET** (ActiveX Data Objects) is a low-level .NET library used to interact directly with databases using SQL commands).

- It's lightweight and efficient.
- Provides direct control over SQL operations.
- You manage SQL queries manually (like SELECT, INSERT, etc.).

Why use ADO.NET in this project?
We use it to perform raw database operations (CRUD) with SQL Server in a simple and controlled manner.

It provides:

- `SqlConnection` → Connect to the DB
- `SqlCommand` → Write SQL queries (INSERT, SELECT, etc.)
- `SqlDataReader` → Read data row-by-row
- `SqlParameter` → Prevent SQL injection
- `ExecuteNonQuery`, `ExecuteReader`, etc.

✅ So how is DAL related to ADO.NET?

When you build a DAL using ADO.NET, you:

1. Create a class (like `EmployeeDataAccess`)
2. Write methods (`GetAll()`, `Add()`, `Update()`, etc.)
3. Use **ADO.NET objects** inside those methods
   - `SqlConnection`, `SqlCommand`, `SqlDataReader`, etc.

So yes — **your DAL is where ADO.NET code lives** and executes!

🛠️ Setup Before Writing Code:

1. ✅ **Create a `Data` folder** in your project root.
   - Right-click project > Add > New Folder > Name it: `Data`
2. ✅ **Create a class** inside `Data` folder.
   - Right-click `Data` > Add > Class > Name it: `EmployeeDataAccess.cs`

📦 Required NuGet Package

Open **NuGet Package Manager Console** (Tools > NuGet Package Manager > Package Manager Console), then run:

```
Install-Package Microsoft.Data.SqlClient
```

Or: Right-click on the project > Manage NuGet Packages > Search and install:
📦 `Microsoft.Data.SqlClient`

✅ EmployeeDataAccess.cs Code with Comments:

```
using Microsoft.Data.SqlClient;          // For SQL Connection
using CrudApiDemo.Models;               // To access Employee model
using System.Data;                      // For Data-related types

namespace CrudApiDemo.Data
{
    public class EmployeeDataAccess
    {
        private readonly string _connectionString;

        // Constructor to get connection string from appsettings.json
        public EmployeeDataAccess(IConfiguration configuration)
        {
            _connectionString = configuration.GetConnectionString("DefaultConnection");
        }

        // ✅ GET All Employees
        public List<Employee> GetAll()
        {
            List<Employee> list = new List<Employee>();
            using (SqlConnection con = new SqlConnection(_connectionString))
            {
                SqlCommand cmd = new SqlCommand("SELECT * FROM Employees", con);
                con.Open();
                SqlDataReader rdr = cmd.ExecuteReader();
                while (rdr.Read())
                {
                    list.Add(new Employee
                    {
                        Id = Convert.ToInt32(rdr["Id"]),
                        Name = rdr["Name"].ToString(),
                        Department = rdr["Department"].ToString(),
                        Salary = Convert.ToInt32(rdr["Salary"])
                    });
                }
            }
            return list;
        }

        // ✅ GET Single Employee by Id
        public Employee Get(int id)
        {
            Employee emp = null;
            using (SqlConnection con = new SqlConnection(_connectionString))
            {
                SqlCommand cmd = new SqlCommand("SELECT * FROM Employees WHERE Id=@Id",
con);
                cmd.Parameters.AddWithValue("@Id", id);
                con.Open();
                SqlDataReader rdr = cmd.ExecuteReader();
                if (rdr.Read())
                {
                    emp = new Employee
                    {
                        Id = Convert.ToInt32(rdr["Id"]),
                        Name = rdr["Name"].ToString(),
                        Department = rdr["Department"].ToString(),
                        Salary = Convert.ToInt32(rdr["Salary"])
                    };
                }
            }
            return emp;
        }

        // ✅ ADD New Employee
        public void Add(Employee emp)
        {
```

```
            using (SqlConnection con = new SqlConnection(_connectionString))
            {
                SqlCommand cmd = new SqlCommand("INSERT INTO Employees (Name,
Department, Salary) VALUES (@Name, @Department, @Salary)", con);
                cmd.Parameters.AddWithValue("@Name", emp.Name);
                cmd.Parameters.AddWithValue("@Department", emp.Department);
                cmd.Parameters.AddWithValue("@Salary", emp.Salary);
                con.Open();
                cmd.ExecuteNonQuery();
            }
        }

        // ✅ UPDATE Employee
        public void Update(Employee emp)
        {
            using (SqlConnection con = new SqlConnection(_connectionString))
            {
                SqlCommand cmd = new SqlCommand("UPDATE Employees SET Name=@Name,
Department=@Department, Salary=@Salary WHERE Id=@Id", con);
                cmd.Parameters.AddWithValue("@Id", emp.Id);
                cmd.Parameters.AddWithValue("@Name", emp.Name);
                cmd.Parameters.AddWithValue("@Department", emp.Department);
                cmd.Parameters.AddWithValue("@Salary", emp.Salary);
                con.Open();
                cmd.ExecuteNonQuery();
            }
        }

        // ✅ DELETE Employee
        public void Delete(int id)
        {
            using (SqlConnection con = new SqlConnection(_connectionString))
            {
                SqlCommand cmd = new SqlCommand("DELETE FROM Employees WHERE Id=@Id",
con);
                cmd.Parameters.AddWithValue("@Id", id);
                con.Open();
                cmd.ExecuteNonQuery();
            }
        }
    }
}
```

✅ These steps ensure your **Model and Data Layer** are fully ready.

✅ Step 7: Create the Controller

🎯 What is a Controller in ASP.NET Core?

- A **Controller** is a class that handles incoming **HTTP requests**, processes them (usually by calling business logic/data access), and returns an appropriate **HTTP response**.
- Controllers are the **heart of the API** — they **map API routes to C# methods** (called action methods).
- Each method in the controller represents a specific HTTP verb (**GET**, **POST**, **PUT**, **DELETE**) and performs corresponding logic (like reading, inserting, updating, deleting data).

📌 Why Do We Need a Controller?

- To handle API routes like `/api/employee`
- To perform logic for each CRUD operation
- To act as a bridge between the client (Postman or frontend) and the data (SQL Server via ADO.NET)

🛠️ How to Create a Controller in Visual Studio

1. **Right-click** on the `Controllers` folder (create it if not already present).
2. Click **Add > Controller**
3. Choose **API > API Controller - Empty**, then click **Add**
4. Name it `EmployeeController.cs`

✅ If `Controllers` folder doesn't exist:

- Right-click the project > Add > New Folder > Name it `Controllers`
- Then follow the steps above to add a new controller inside it.

💡 Controller Code with Comments

```
using Microsoft.AspNetCore.Mvc;          // For API Controller and attributes
using CrudApiDemo.Models;                // To use the Employee model
using CrudApiDemo.Data;                  // To use the EmployeeDataAccess class

namespace CrudApiDemo.Controllers
{
    // Defines the route pattern: api/employee
    [Route("api/[controller]")]
    [ApiController] // Enables API-specific features like automatic model validation
    public class EmployeeController : ControllerBase
    {
        private readonly EmployeeDataAccess _data;

        // Constructor: Inject configuration to access connection string
        public EmployeeController(IConfiguration config)
        {
            _data = new EmployeeDataAccess(config);
        }

        // GET: api/employee
        [HttpGet]
        public IActionResult GetAll() => Ok(_data.GetAll());

        // GET: api/employee/1
        [HttpGet("{id}")]
        public IActionResult Get(int id)
        {
            var emp = _data.Get(id);
            if (emp == null)
                return NotFound(); // 404 if not found
            return Ok(emp);        // 200 with employee data
        }

        // POST: api/employee
        [HttpPost]
        public IActionResult Post(Employee emp)
        {
            _data.Add(emp);
            return Ok("Added");    // 200 OK after adding
```

```
        }

        // PUT: api/employee
        [HttpPut]
        public IActionResult Put(Employee emp)
        {
            _data.Update(emp);
            return Ok("Updated");  // 200 OK after updating
        }

        // DELETE: api/employee/1
        [HttpDelete("{id}")]
        public IActionResult Delete(int id)
        {
            _data.Delete(id);
            return Ok("Deleted");  // 200 OK after deletion
        }
    }
}
```

▶️ Now Run the API:

1. Press `Ctrl + F5` or click **Start Without Debugging**
2. Your app should launch and show `Now listening on: http://localhost:XXXX`

✅ Step 8: Test Your API Using Postman

🔷 1. Insert Data into the Table

---

a. Using POST Method in Postman

- **URL**: `http://localhost:5021/api/employee`
- **Method** : `POST`
- **Go to** : `Body → raw → JSON`
- **Paste JSON**:

```
{
  "name": "Sagar",
  "department": "IT",
  "salary": 60000
}
```

- **Click**:
  `Send`

✅ You should get:

```
"Added"
```

---

b. Using SSMS (SQL Server Management Studio)

- Connect to your server
- Open a New Query window in `PracticeDb`
- Paste and run this SQL:

```sql
USE PracticeDB;

INSERT INTO Employees (Name, Department, Salary)
VALUES
('Sagar Haldar', 'IT', 55000),
('Anjali Singh', 'HR', 48000),
('Rahul Sharma', 'Finance', 60000);


Select * from Employees
```

✅ Run this by pressing `F5` or the **Execute** button.

🔷 2. Get the Data (GET All Employees)

- **URL**:

  `http://localhost:5021/api/employee`

- **Method**:
  `GET`
- ✅ Click `Send`

Expected Response:

```
[
  {
    "id": 1,
    "name": "Sagar",
    "department": "IT",
    "salary": 60000
  },
  {
    "id": 2,
    "name": "Anjali",
    "department": "HR",
    "salary": 50000
  }
]
```

🔷 3. Update Data using PUT Method

- **URL**:

  ```
  http://localhost:5021/api/employee
  ```

- **Method**:
  `PUT`
- **Go to**:
  `Body → raw → JSON`
- **Paste JSON** (change details as needed):

  ```
  {
  ```

```
  "id": 1,
  "name": "Sagar H",
  "department": "Development",
  "salary": 70000
}
```

- ✅ Click `Send`

Expected:

```
"Updated"
```

🔷 4. Delete Data using DELETE Method

- **URL**:

  ```
  http://localhost:5021/api/employee/2
  ```

  here ‘2’ is the id, whose Data we want to Delete.

- **Method**:
  `DELETE`
- ✅ Click `Send`

Expected:

```
"Deleted"
```

✅ Postman Status Recap

Operation	Method	URL	Body Required?	Purpose
Insert (POST)	POST	`http://localhost:5021/api/employee`	✅ Yes	Add new employee
Read All	GET	`http://localhost:5021/api/employee`	❌ No	Get all employees
Read by ID	GET	`http://localhost:5021/api/employee/1`	❌ No	Get employee by ID
Update (PUT)	PUT	`http://localhost:5021/api/employee`	✅ Yes	Update employee
Delete (DELETE)	DELETE	`http://localhost:5021/api/employee/2`	❌ No	Delete employee

WORKING WITH GITHUB

✅ PART 1: Upload an ASP.NET Core Web API Project to GitHub (Initial Setup)

🔷 Step 1: Create a GitHub Repository

1. Go to https://github.com
2. Click **New Repository**
3. Fill in:
   - **Name**: `CrudApiDemo` (or your project name)
   - **Description** (optional)
   - Keep it **Public** or **Private**
   - Uncheck: `Initialize this repository with a README`
4. Click **Create Repository**

🔷 Step 2: Push Your ASP.NET Core Project to GitHub from Visual Studio

1. Open your project (`CrudApiDemo`) in **Visual Studio**
2. Go to **View > Git Changes**
3. Click **"Initialize Repository"** (if not already initialized)
4. Now go to **Git Changes pane**:
   - Stage All Changes
   - Add a Commit Message like: `Initial commit - ASP.NET Core Web API CRUD`
5. Click **"Push to GitHub"** or set the remote manually:
   - In Terminal:

```
git remote add origin https://github.com/YourUsername/CrudApiDemo.git
git branch -M main
git push -u origin main
```

🔁 You can also go to **Git > Manage Remotes** in Visual Studio and paste your repo URL.

🔷 Step 3: Add `.gitignore` and `README.md`

1. Add a **`.gitignore`** (use gitignore.io for template: choose `.NET`, `VisualStudio`)
2. Add a **`README.md`** to explain:
   - Project overview
   - Tech used: ASP.NET Core, ADO.NET, SQL Server
   - How to run locally

✅ PART 2: Clone and Work with an Existing Repo (Team Environment)

🔷 Step 1: Clone a GitHub Repository using Visual Studio

1. In Visual Studio: `File → Clone Repository`
2. Paste the URL (from GitHub: `Code > HTTPS`)
3. Choose your local folder (e.g., `D:\TeamProjects`)
4. Click **Clone**

✅ This will download the project to your system and open it directly in Visual Studio.

🔷 Step 2: Handle Database (SQL Server)

💡 **Databases are not stored on GitHub**, so teammates need to **recreate them locally**.

To set up DB:

- Use a shared `.sql` file (DB schema + seed data)
- OR, write setup instructions in `README.md`

Example SQL:

```
CREATE DATABASE PracticeDb;

USE PracticeDb;

CREATE TABLE Employees (
    Id INT PRIMARY KEY IDENTITY,
    Name NVARCHAR(100),
    Department NVARCHAR(50),
    Salary INT
);
```

You can also add a folder like `Docs/SQL-Scripts/InitialSchema.sql`.

✅ PART 3: Work as a Team and Push Code

🔷 Step 1: Make Code Changes

- Pull latest code first:
  - In Visual Studio: **Git > Pull**
- Make your changes
- Build and run the app locally to verify

🔷 Step 2: Stage → Commit → Push

1. Go to **Git Changes**
2. Stage modified files
3. Write a clear commit message:

   ```
   Added EmployeeController and completed CRUD operations
   ```

4. Click **Commit All and Push**

🔷 Step 3: Pull Regularly

To avoid **merge conflicts**, always:

```
git pull origin main
```

before you start your day's work.

🔷 Step 4: Branching (Optional but Recommended in Team Projects)

1. **Create a new branch** for every feature:

   ```
   git checkout -b feature/add-logging
   ```

2. Make changes → Commit → Push to that branch
3. Create a **Pull Request (PR)** on GitHub
4. Teammates review and merge to main

✅ Summary of Required Professional Practices

Task	Tool/Action
Code Versioning	Git + GitHub
API Testing	Postman
Database	SQL Server + SSMS
Remote Setup	Clone Repo, Set Connection Strings
Syncing Work	Git Pull / Push / Branches
Project Docs	Add `README.md`, `.gitignore`, `SQL Scripts` folder