API Development with C# .NET

1. Introduction
   1. This courses covers C#, .NET and MS-SQL.
   2. Requirements
      1. Microsoft .NET SDK 9.0.303
      2. Visual Studio
2. Quick Links to Detailed Subjects

* 1. [GitHub Account info](#S01)

* 1. [C# Basics](#S02)
     1. [Connecting to Nuget](#S02_01)
     2. [Creating a console app](#S02_02)
     3. [Different types of variables](#S02_03)
     4. [Data Structures](#S02_04)
        1. [Arrays](#S02_04_01)
        2. [Lists](#S02_04_02)
        3. [Dictionaries](#S02_04_03)
     5. [Operators](#S02_05)
     6. [Conditional Statements](#S02_06)
  2. Intermediate Part I
     1. [Models](#S03_01)
     2. [Namespaces](#S03_02)
     3. [Dapper Definition and Planning](#S03_03)
     4. [Entity Framework Introduction](#S03_04)
     5. [Examples Using Entity Framework](#S03_05)
     6. [Config File](#S03_06)
  3. Intermediate Part II
     1. [Files](#S04_01)
     2. [JSON](#S04_02)
     3. [Model Mapping](#S04_03)
     4. [Async and Await](#S04_04)
  4. SQL Intermediate
     1. [Backing up and restoring a database](#S05_01)
     2. [Joins](#S05_02)
     3. [JOIN vs WHERE](#S05_03)
     4. [Creating Indexes](#S05_04)
     5. [Outer Apply and Cross Apply](#S05_05)
     6. [Date Examples](#S05_06)
     7. [Alter Table](#S05_07)

* 1. [API Basics](#S06)
     1. [Introduction](#S06_01)
     2. [Installing Swagger](#S06_02)
     3. [Starting Swagger and Creating a Controller](#S06_03)
     4. [Setting Up a Controller](#S06_04)
     5. [Creating our First Custom Controller](#S06_05)
     6. [Shutdown and Restart the Application](#S06_06)
     7. [Updating the LaunchSettings.json file](#S06_07)

* 1. [CORS](#S07)
     1. [Introduction](#S07_01)

* 1. [Database Connections](#S08)

* 1. [User Models](#S09)

* 1. [User Controller](#S10)
     1. [The QS Class](#S10_01)
     2. [HttpGet](#S10_02)
     3. [HttpPut and HttpPost](#S10_03)
     4. [HttpDelete](#S10_04)

* 1. [DTO](#S11)

* 1. [Namespaces](#S12)

* 1. [Entity Framework](#S13) [Setup](#S13)

* 1. [Entity Framework Controller](#S14)

* 1. [The User Repository and User Repository Interface](#S15)

1. [GitHub Account info](#TS01):
   1. User Name: DaveJaynes
   2. Password: VassarMichigan48768
   3. Repository: Udemy
   4. Example of creating a new repository:
      1. echo "# Udemy" >> README.md
      2. git init
      3. git add README.md
      4. git commit -m "first commit"
      5. git branch -M main
      6. git remote add origin https://github.com/DaveJaynes/Udemy.git
      7. git push -u origin main
   5. Example of pushing an existing repository from the command line
      1. git remote add origin https://github.com/DaveJaynes/Udemy.git
      2. git branch -M main
      3. git push -u origin main

1. [C# Basics](#TS02)

* 1. [Connecting to Nuget](#TS02)
     1. Type these lines in PowerShell:
        1. dotnet nuget list source
        2. dotnet nuget add source https://api.nuget.org/v3/index.json -n nuget.org

* 1. [Creating a console app](#TS02)
     1. Open PowerShell
     2. Create a new dotnet app
        1. dotnet new console -n HelloWorld
     3. This will create a new directory with the name HelloWorld
        1. The console application lives in this directory
     4. In this directory will be two files and a directory:
        1. HelloWorld.csproj
        2. Program.cs
        3. The obj directory
     5. To run this program:
        1. Change directory to HelloWorld
        2. Inside the HelloWorld directory, we can run the program by typing: dotnet run
        3. This will output: Hello, World!
     6. We can now type: code .
        1. This will open Visual Studio Code with our current project.
        2. To open VSC, type this at the PS prompt inside the new folder generated by the dotnet new console command: **code .**

* 1. [Different types of variables](#TS02)
     1. All these types are static.
     2. You can write these line to get their types:
        1. Console.WriteLine(myString.GetType());
        2. Console.WriteLine(myDecimal.GetType());
        3. ConsoleWriteLine(myBoolean.GetType());

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Bytes Used | Value Range | Description |
| byte | 1 | 0 to 255 | Unsigned byte |
| sbyte | 1 | -128 to 127 | Signed byte |
| ushort | 2 | 0 to 65535 | Unsigned short |
| short | 2 | -32768 to 32767 | Signed short |
| int | 4 | -2147483648 to 2147483648 | Integer |
| long | 8 | -9223372036854775808 to 9223372036854775807 | Long |
| float | 4 bytes | 12.45f | Just like in Java, a float number needs to end with the letter F. |
| double | 8 bytes | 12.45 |  |
| decimal | 16 bytes | 12.45m | Decimal numbers need to end with the letter M. |
| char | 2 bytes |  | Must be surrounded by a single quote |
| string | Depends |  | Must be surrounded by double quotes |
| bool | 1 |  | True or False |

* 1. [Data Structures](#TS02)

* + 1. [Arrays](#TS02)
       1. When you declare an array, its size is fixed.
       2. The script below explains how to create string arrays. It works the same as other data types.

Example of Creating Arrays in C#

|  |
| --- |
| using System;  namespace HelloWorld;  internal class Program  {  static void Main(string[] args)  {  // First way to declare and initialize an array  string[] myArray = new string[2];  myArray[0] = “Hello, World!”;  myArray[1] = “Welcome to C# programming.”;  foreach (string item in myArray)  {  Console.WriteLine(item);  }  // Second way to declare and initialize an array  string[] greetings = { “Hello, World!”, “Welcome to C# programming.”, “Enjoy coding!” };  foreach (string greeting in greetings)  {  Console.WriteLine(greeting);  }  }  } |

* + 1. [Lists](#TS02)
       1. Just line in Java, List elements can grow on the fly.
       2. Below we have a script that displays the use of lists, Enumerable Lists, and Multi-Dimensional arrays

Examples of List Enumerable List and Multi-Dimensional Arrays

|  |
| --- |
| using System;  namespace HelloWorld;  internal class Program  {  static void Main(string[] args)  {  List<string> myGroceryList = new List<string>();  myGroceryList.Add("Apples");  myGroceryList.Add("Bananas");  myGroceryList.Add("Carrots");  foreach (string item in myGroceryList)  {  Console.WriteLine(item);  }  // We can simplify the list initialization using a collection initializer.  List<int> myNumbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  // IEnumerable exposes the enumerator, which supports a simple iteration over a collection of a specified type.  // Here the result would be 2,4,6,8,10  IEnumerable<int> myEvenNumbers = myNumbers.Where(n => n % 2 == 0);  foreach (int number in myEvenNumbers)  {  Console.Write(number + ", ");  }  // Finally, we can convert the IEnumerable back to a List.  List<int> evenNumbersList = myEvenNumbers.ToList();  // Multi-dimensional arrays can be used to store data in a grid-like structure.  int[,] grid = new int[3, 3]  {  { 1, 2, 3 },  { 4, 5, 6 },  { 7, 8, 9 }  };  // Displaying the grid  for (int i = 0; i < grid.GetLength(0); i++)  {  for (int j = 0; j < grid.GetLength(1); j++)  {  Console.Write(grid[i, j] + " ");  }  Console.WriteLine();  }  }  } |

* + - * 1. This is the output of the program:

|  |
| --- |
| Apples  Bananas  Carrots  2, 4, 6, 8, 10, 1 2 3  4 5 6  7 8 9 |

* + 1. [Dictionaries](#TS02)
       1. These are Key/Value pairs.
       2. You access the data in the dictionary by searching for the key value.
       3. Below is a script that explains some common Dictionary commands:

Example of Using Dictionaries

|  |
| --- |
| using System;  namespace Dictionaries;  internal class Program  {  static void Main(string[] args)  {  Dictionary<string, int> groceryPrices = new Dictionary<string, int>();  groceryPrices.Add("Apple", 1);  groceryPrices.Add("Banana", 2);  groceryPrices.Add("Orange", 3);  groceryPrices.Add("Mango", 4);  groceryPrices.Add("Grapes", 5);  Console.WriteLine("Grocery Prices:");  foreach (var item in groceryPrices)  {  Console.WriteLine($"{item.Key}: ${item.Value}");  }  // Accessing the price of Banana  Console.WriteLine(groceryPrices["Banana"]);  // Update the price of Banana  groceryPrices["Banana"] = 3;  Console.WriteLine($"Updated Banana Price: ${groceryPrices["Banana"]}");  // Remove "Orange" from the dictionary  groceryPrices.Remove("Orange");    // Check to see if "Apple" is in the dictionary  Console.WriteLine(groceryPrices.ContainsKey("Apple") ? "Apple is available." : "Apple is not available.");  }  } |

* + - * 1. This is the output of the program:

|  |
| --- |
| Grocery Prices:  Apple: $1  Banana: $2  Orange: $3  Mango: $4  Grapes: $5  2  Updated Banana Price: $3  Apple is available. |

* 1. [Operators](#TS02)
     1. This is the basic less-than, greater than type commands.
     2. Below is an example script

Example of Using Operators

|  |
| --- |
| using System;  namespace Operators;  class Program  {  static void Main(string[] args)  {  // Splitting string text.  // Note: To suppress CS8600 warnings, we can use the NoWarn property <NoWarn>CS8600</NoWarn> in the .csproj file.  Console.Write("Enter a two part phrase: ");  string input = Console.ReadLine();  if (string.IsNullOrEmpty(input))  {  Console.WriteLine("Input cannot be empty.");  return;  }  else  {  string[] splitingInput = input.Split(" ");  string firstString = splitingInput[0];  string secondString = splitingInput[1];  string result = firstString + secondString;  Console.WriteLine(result);  Console.WriteLine(firstString + " " + secondString);  Console.WriteLine($"{firstString} {secondString}");  }  // Comparing values  int a = 10;  int b = 20;  bool result1 = a.Equals(b);  if (result1) { Console.WriteLine("a is equal to b"); } else { Console.WriteLine("a is not equal to b"); }  // We could also use the == operator: bool result1 = a == b;    result1 = a.Equals(b/2);  if (result1) { Console.WriteLine("a is equal to b"); } else { Console.WriteLine("a is not equal to b"); }  }  } |

* 1. [Conditional Statements](#TS02)
     1. For loop conditional statements

Example of For and ForEach Loops

|  |
| --- |
| using System;  namespace Conditions;  internal class Program  {  private static void Main(string[] args)  {  Console.WriteLine("\n\nSpeed comparisons between adding elements independently, using for loop and using foreach loop");  Console.WriteLine("==============================================================================================\n");  int[] integers\_to\_add = new int[] { 10, 15, 20, 25, 30, 12, 34 };  // Run one  Console.WriteLine("Run 1: Just adding all the array elements together");  Console.WriteLine("--------------------------------------------------");  DateTime startTime1 = DateTime.Now;  int totalValue = integers\_to\_add[0] + integers\_to\_add[1] + integers\_to\_add[2] + integers\_to\_add[3] + integers\_to\_add[4] + integers\_to\_add[5] + integers\_to\_add[6];  DateTime endTime1 = DateTime.Now;  double totalTime1 = (endTime1 - startTime1).TotalMilliseconds;  Console.WriteLine("Total time for run 1 is " + totalTime1 + " MilliSeconds.\n");  // Run two  Console.WriteLine("Run 2: Adding array elements together in for loop");  Console.WriteLine("-------------------------------------------------");  DateTime startTime2 = DateTime.Now;  totalValue = 0;  for (int i = 0; i < integers\_to\_add.Length; i++)  {  totalValue += integers\_to\_add[i];  }  DateTime endTime2 = DateTime.Now;  double totalTime2 = (endTime2 - startTime2).TotalMilliseconds;  Console.WriteLine("Total time for run 2 is " + totalTime2 + " MilliSeconds.\n");  // Run three  Console.WriteLine("Run 3: Adding array elements together in for-each loop");  Console.WriteLine("------------------------------------------------------");  DateTime startTime3 = DateTime.Now;  totalValue = 0;  foreach (int each\_int in integers\_to\_add)  totalValue += each\_int;  DateTime endTime3 = DateTime.Now;  double totalTime3 = (endTime3 - startTime3).TotalMilliseconds;  Console.WriteLine("Total time for run 3 is " + totalTime3 + " MilliSeconds.\n\n");  }  } |

* + 1. While and Do-While Loop Statements

Example of While and D0-While Statements

|  |
| --- |
| namespace Conditions;  public class WhileLoop  {  DateTime startTime;  DateTime endTime;  double totalTime;  int index;  int totalValue;  int[] integers\_to\_add = new int[] { 10, 15, 20, 25, 30, 12, 34 };  public void Heading()  {  Console.WriteLine("\n\nSpeed comparisons between while and do while loops");  Console.WriteLine("====================================================\n");  }  public void whileloop()  {  // Run one  Console.WriteLine("Run 1: Performing while loop");  Console.WriteLine("----------------------------");  index = 0;  totalValue = 0;  startTime = DateTime.Now;  while(index < integers\_to\_add.Length)  {  totalValue += integers\_to\_add[index];  index++;  }  endTime = DateTime.Now;  totalTime = (endTime - startTime).TotalMilliseconds;  Console.WriteLine("Total time for run 1 is " + totalTime + " MilliSeconds.\n");  }  public void dowhileloop()  {  // Run one  Console.WriteLine("Run 2: Performing do-while loop");  Console.WriteLine("-------------------------------");  index = 0;  totalValue = 0;  startTime = DateTime.Now;  do  {  totalValue += integers\_to\_add[index];  index++;  }while(index < integers\_to\_add.Length);  endTime = DateTime.Now;  totalTime = (endTime - startTime).TotalMilliseconds;  Console.WriteLine("Total time for run 2 is " + totalTime + " MilliSeconds.\n");  return;  }  } |

1. [SQL Intermediate Part I](#TS03)

* 1. [Models](#TS03)
     1. A way to map data from one place to another, for example from SQL to our application.
     2. Models are nothing more than creating your own class and assigning it properties.
     3. Let’s take a look at a script called Program.cs that both creates a Model (a Class) and then instantiates it and instance assigns *values* to that model.

Example of Creating a Model and Instantiating It in our Program.cs Program

|  |
| --- |
| using System;  using System.Text.RegularExpressions;  namespace Programs  {  public class Computer  {  // {get; set;} creates a private anonymous motherboard field in the background.  // The  = ""; after Motherboard and VideoCard automatically assign it a non-null value.  public string Motherboard { get; set; }  = "";  public int CPUCores { get; set; }  public bool HasWifi { get; set; }  public bool HasLTE { get; set; }  public DateTime ReleaseDate { get; set; }  public decimal Price { get; set; }  public string VideoCard { get; set; }  = "";  }  internal class Program  {  static void Main(string[] args)  {  Computer myComputer = new Computer()  {  Motherboard = "Z690",  HasWifi = true,  HasLTE = false,  ReleaseDate = DateTime.Now,  Price = 943.87m,  VideoCard = "RTX 2060"  };  Console.WriteLine(myComputer.Motherboard);  Console.WriteLine(myComputer.HasWifi);  Console.WriteLine(myComputer.ReleaseDate);  Console.WriteLine(myComputer.VideoCard);  }  }  } |

* 1. [Namespaces](#TS03)
     1. Referencing files in different *folders*, which are referred to as *Namespaces*.
        1. It is often a popular practice of creating a new class within a separate file outside our own Program.cs file.
        2. For instance, in the example above, let’s assume the Program.cs file resides in the folder C:\CSharp\Applications\Programs.
           1. Notice the namespace for our example script is Programs as that is the relative starting point of executing within the computers directory structure.
        3. We could have just as easily created a new file called Computer.cs and put all of that classes code in that file.
        4. This would just file if the new Computer.cs file is in the same folder (Namespace) as the Program.cs file is in.
        5. But what is the new Class file we create is in a separate folder located at C:\CSharp\Applications\Programs\Models.
           1. Now the programs Program.cs and Computer.cs can no longer see each other because they reside in different folders, or more accurately different *Namespaces*.
        6. Resolving this issue.
           1. To ensure these two files can once again reference each other, we need to add a few lines of code in each file.
           2. In the new Computer.cs file:

We will add a namespace line that lists its location relative to the Program.cs file that will be calling it.

That namespace line will be: namespace Computers.Models {}

* + - * 1. In our original Program.cs file, we will need to import the Computers.Models directory with this command: using Computers.Models;
        2. Now the two files have links to each other and the world is safe again.
      1. Below are the two scripts in full:

Displaying the Program.cs and Computer.cs files

Program.cs File

|  |
| --- |
| using System;  using System.Text.RegularExpressions;  using Computers.Models;  namespace Computers  {  internal class Program  {  static void Main(string[] args)  {  Computer myComputer = new Computer()  {  Motherboard = "Z690",  HasWifi = true,  HasLTE = false,  ReleaseDate = DateTime.Now,  Price = 943.87m,  VideoCard = "RTX 2060"  };  Console.WriteLine(myComputer.Motherboard);  Console.WriteLine(myComputer.HasWifi);  Console.WriteLine(myComputer.ReleaseDate);  Console.WriteLine(myComputer.VideoCard);  }  }  } |

Computers.cs File

|  |
| --- |
| namespace Computers.Models  {  public class Computer  {  // Get/Set Creates a private anonymous motherboard field in the background.  public string Motherboard { get; set; } = "";  public int CPUCores { get; set; }  public bool HasWifi { get; set; }  public bool HasLTE { get; set; }  public DateTime ReleaseDate { get; set; }  public decimal Price { get; set; }  public string VideoCard { get; set; } = "";  }  } |

* 1. [Dapper Definitions and Planning](#TS03)
     1. Dapper is a tool used to connect to the database and run sequel queries.
     2. It uses IDbConnection and SqlConnection.
        1. Just fyi, IdbConnection is an Interface and SqlConnection is a class.
     3. Types of connection requests:
        1. Query/QuerySingle
        2. Execute
     4. Adding new .NET packages to allow database connection:
        1. Inside a PowerShell window, issue these four commands. (You should probably ensure you have admin rights):
           1. dotnet add package Dapper
           2. dotnet add package microsoft.data.sqlclient
           3. dotnet add package microsoft.entityframeworkcore
           4. dotnet add package microsoft.entityframeworkcore.sqlserver
     5. Close down Visual Studio Code and restart it with the code . command.
        1. Just as a note, you will still be in the Computers directory where we left off last while running the namespace programs.
     6. Now let’s verify these packages were loaded successfully:
        1. Once Visual Studio Code opens, open up the Computers.csproj file and look for the code highlighted below:

|  |
| --- |
| <Project Sdk="Microsoft.NET.Sdk">  <PropertyGroup>  <OutputType>Exe</OutputType>  <TargetFramework>net9.0</TargetFramework>  <ImplicitUsings>enable</ImplicitUsings>  <Nullable>enable</Nullable>  </PropertyGroup>  <ItemGroup>  <PackageReference Include="Dapper" Version="2.1.66" />  <PackageReference Include="Microsoft.Data.SqlClient" Version="6.1.0" />  <PackageReference Include="Microsoft.EntityFrameworkCore" Version="9.0.7" />  <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="9.0.7" />  </ItemGroup>  </Project> |

* + - 1. If you see these highlighted lines in the Computers.csproj file, you are good to go.
    1. Now lets create a connection string variable in our program.
       1. Our connection string using a Windows Trusted Connection would be:
          1. string connectionString = "Server=DAVES\_PC;Database=DotNetCourseDatabase;TrustServerCertificate=true;Trusted\_Connection=true;";
       2. Cour connection string using a username and password would be:
          1. string connectionString = "Server=DAVES\_PC;Database=DotNetCourseDatabase;TrustServerCertificate=true;Trusted\_Connection=false;User ID=*myusername*;Password=*mypassword*;”;
       3. Next, we use the IdbConnection command:
          1. IDbConnection dbConnection = new SqlConnection(connectionString);
       4. Now we will test our connection with these three lines:
          1. The first line request the date from SQL server.
          2. The second line uses QuerySingle to get only one response.

Notice it is typeset as DateTime and uses the sqlCommand as a parameter.

* + - * 1. Finally, we display our output to the console.
        2. Below are all the lines of code need to set up the DB connection and query the system date:

|  |
| --- |
| string connectionString = "Server=DAVES\_PC;Database=DotNetCourseDatabase;TrustServerCertificate=true;Trusted\_Connection=true;";  IDbConnection dbConnection = new SqlConnection(connectionString);  string sqlCommand = “SELECT GETDATE()”;  DateTime rightNow = dbConnection.QuerySingle<DateTime>(sqlCommand);  Console.WriteLine(“The DateTime right now is [“ + rightNow + “]”); |

* + - 1. If our connection ran successfully, this will be our output:

|  |
| --- |
| The DateTime right now is [7/26/2025 6:07:23 PM] |

* + 1. Dapper non query SQL commands
       1. We use the Execute command to perform non-query commands.
       2. The code below displays how we can insert data from on entry made in the Computer class.

Entering data into the Computer table which was extracted from the object of the Computer Class

|  |
| --- |
| string connectionString = "Server=DAVES\_PC;Database=DotNetCourseDatabase;TrustServerCertificate=true;Trusted\_Connection=true;";  IDbConnection dbConnection = new SqlConnection(connectionString);  // Add some data to the object of the Computer class.  Computer myComputer = new Computer()  {  Motherboard = "Z690",  HasWifi = true,  HasLTE = false,  ReleaseDate = DateTime.Now,  Price = 943.87m,  VideoCard = "RTX 2060"  };  // Inserting a row into the TutorialAppSchema.Computer table extracted from the object of the Computer class..  // Note: The @ symbol allows us to enter multiple lines of text into the same command.  string sql = @"insert into TutorialAppSchema.Computer(motherboard,hasWifi,hasLTE,releaseDate,price,videoCard)  values('" + myComputer.Motherboard  + "','" + myComputer.HasWifi  + "','" + myComputer.HasLTE  + "','" + myComputer.ReleaseDate  + "','" + myComputer.Price  + "','" + myComputer.VideoCard  + "')";  // The number return by this statement will be the number of rows inserted. Here we check if that number is greater than zero.  if (dbConnection.Execute(sql) > 0) Console.WriteLine("Good write"); else Console.WriteLine("Good write"); |

* + 1. Using IEnumerable to return multiple columns of data
       1. When the query method of dbConnection is used, an IEnumerable data type is returns.
       2. For that reason, we need to capture the SQL response with an IEnumerable object.
       3. The command will look like this:

|  |
| --- |
| IEnumerable<Computer> computers = dbConnection.Query<Computer>(sqlSelect); |

* + - 1. IEnumerable works like an escort for a group of item or objects.
      2. It gathers them together like a tour guide and escorts them one by one to a process such as a foreach loop.
      3. It this case, when all the rows of the database table are read, each row is instantiated into their own Computer class object.
      4. These objects are like railroad cars and IEnumerable is the locomotive engine hauling them to the foreach processing plant.
  1. [Entity Framework Introduction](#TS03)
     1. Individuals who are building out an application and do not know SQL can use Entity Framework because you do not need to know SQL to use it.
        1. Note: LEARN SQL and USE DAPPER INSTEAD!
     2. Again, this is a method to access SQL without having to write SQL code.
     3. We will be using DbContext and inheriting from it to use with Entity Framework.
     4. OnConfiguring will work with DbContext and will fire when we create an instance of Entity Framework.
        1. It will override the OnConfiguring method in DbContext.
        2. It will be using UseSqlServer to connect to SQL.
        3. It tells where the database is.
     5. OnModelCreating fires when we create an instance of a model.
        1. It will override the OnModelCreating method within DbContext.
        2. It tells where the table is inside the database.
     6. DBSet / Entity
        1. Entity tells DbSet how to get to the table.
     7. Interactions
        1. This allows for:
           1. Adding added using DbContext.Model.Add
           2. Saving data using DbContext.SaveChanges
           3. Querying data using DbContext.Model.ToList

* 1. [Examples using Entity Framework](#TS03)
     1. Building an ***Insert*** Entity Framework process with the Program.cs and DataContextEF.cs scripts.
        1. Program.cs
           1. Contains the main entry method.
           2. Instantiates the DataContextEF object.
           3. Instantiate Computer class object with preset values
           4. Using the EF Data Context object, we execute the Add and SaveChanges methods.
        2. DataContextEF.cs
           1. Extends the DBContext class.

We will be overriding five of the DBContext methods.

OnConfiguring(DbContextOptionsBuilder options)

options.UseSqlServer(connString, options => options.EnableRetryOnFailure());

modelBuilder.HasDefaultSchema("TutorialAppSchema");

modelBuilder.Entity<Computer>();

* + - 1. Below are the two scripts:

Program.cs

|  |
| --- |
| using Computers.Models;  using Computers.Data;  namespace Computers  {  internal class Program  {  public static void Main(string[] args)  {  // Set up data context using Entity Framework  DataContextEF entityFramework = new DataContextEF();  // Instantiate Computer class object with preset values.  Computer myComputer = new Computer()  {  Id = 1,  Motherboard = "Z690",  HasWifi = true,  HasLTE = false,  ReleaseDate = DateTime.Now,  Price = 943.87m,  VideoCard = "RTX 2060"  };  // Using the EF Data Context object, execute the Add and SaveChanges methods.  entityFramework.Add(myComputer);  entityFramework.SaveChanges();  }  }  } |

DataContextEF.cs

|  |
| --- |
| using Computers.Models;  using Microsoft.EntityFrameworkCore;  namespace Computers.Data  {  public class DataContextEF : DbContext  {  public DbSet<Computer>? computer { get; set; }  protected override void OnConfiguring(DbContextOptionsBuilder options)  {  // When the DataContextEF class is created, we check to see if it has been configured. if not, we configure it.  if (!options.IsConfigured)  {  string connString = "Server=DAVES\_PC;Database=DotNetCourseDatabase;TrustServerCertificate=true;Trusted\_Connection=true;";  options.UseSqlServer(connString, options => options.EnableRetryOnFailure());  }  }  // ModelBuilder maps an actual model to a table  protected override void OnModelCreating(ModelBuilder modelBuilder)  {  // Set our Schema name  modelBuilder.HasDefaultSchema("TutorialAppSchema");  // Map the entity directly to the 'Computer' table within the 'TutorialAppSchema' schema.  // We use the HasKey keyword to reference the key field in our table.  modelBuilder.Entity<Computer>()  .HasKey(c => c.Id);    /\*  Note: If you are using a table in another schema other than the one set as default,  you will need to use this code instead:  modelBuilder.Entity<Computer>().ToTable("TableName","SchemaName");  \*/  }  }  } |

* + 1. Building a program to query all the rows in the Computer database.
       1. No changes need to be made to the DataContextEF.cs file.
       2. Below is how we change our Program.cs file to display all records using the Entity Frame connection.

Retrieving All Records From the Computer SQL Table using Entity Framework

|  |
| --- |
| using Computers.Models;  using Computers.Data;  namespace Computers  {  internal class Program  {  public static void Main(string[] args)  {  Computer myComputer = new Computer();  // Set up data context using Entity Framework  DataContextEF entityFramework = new DataContextEF();  // Entity Framework will use its connections information pass the SQL data back to IEnumerable.  // No changes need to be made to the DataContextEF.cs file.  IEnumerable<Computer>? computers = entityFramework.computer?.ToList<Computer>();    // Notice eachComputer in the foreach loop? That holds the instance of  // each class, so eachComputer needs to be the reference for the  // Console.WriteLine() commands below as well.  foreach (Computer eachComputer in computers)  {  Console.WriteLine("Motherboard: " + eachComputer.Motherboard);  Console.WriteLine("HasWifi: " + eachComputer.HasWifi);  Console.WriteLine("HasLTE: " + eachComputer.HasLTE);  Console.WriteLine("ReleaseDate: " + eachComputer.ReleaseDate);  Console.WriteLine("Price: " + eachComputer.Price);  Console.WriteLine("VideoCard: " + myComputer.VideoCard);  Console.WriteLine("-------------------------------------------------------");  }  }  }  } |

* 1. [Config file](#TS03)
     1. This stores things like connection strings.
     2. How to set up.
        1. In the same directory as our Program.cs file live, create a file called appsettings.json.
           1. We will see in a few paragraphs how placing it in this directory will cause an issue and how to fix that issue.
        2. Add the following lines to it:

|  |
| --- |
| {  "ConnectionStrings": {  "DefaultConnection":"Server=DAVES\_PC;Database=DotNetCourseDatabase;TrustServerCertificate=true;Trusted\_Connection=true;"  }  } |

* + - 1. In our Program.cs file, add the following line:
         1. using Microsoft.Extensions.Configuration;
         2. IConfiguration config = new ModelConfigurationBuilder();

Make this the first line inside public static void Main.

* + - 1. Close Visual Studio Code
      2. In our PowerShell console, add these two packages:
         1. dotnet add package Microsoft.Extensions.Configuration
         2. dotnet add package Microsoft.Extensions.Configuration.Json
      3. Next, enter the command : dotnet restore
      4. Type code . to relaunch Visual Studio Code
      5. Add this line as the first line inside public static void Main curly braces:
         1. IConfiguration config = new ModelConfigurationBuilder().AddJsonFile(“appsettings.json”).Build();
      6. Whether we are using Dapper or Entity Framework (or both) for our Data Context, we need to add config in the parentheses :
         1. DataContextDapper dapper = new DataContextDapper(config);
         2. DataContextEF entityFramework = new DataContextEF(config);
    1. Updating our DataContextDapper.cs and DataContextEF.cs files with the config file connection.
       1. In both these files, we are going to have to add a private IConfiguration property as well as create the Constructor code to create a private copy of this config.
       2. Update both these files, adding this code between the class line and the first method:

|  |
| --- |
| private IConfiguration \_config;  public DataContextEF(IConfiguration config)  {  \_config = config;  } |

* + - 1. For the DataContextDapper.cs file, we will replace the actual connection string with the \_config GetConnString settings.
      2. For the DataContextEF.cs file, we will replace the connString variable with the \_config GetConnString settings.
      3. Below is the updated DataContextDapper.cs file. The changes will be in blue:

|  |
| --- |
| using System.Data;  using Dapper;  using Microsoft.Data.SqlClient;  using Microsoft.Extensions.Configuration;  namespace Computers.Data  {  public class DataContextDapper  {  private IConfiguration \_config;    public DataContextDapper(IConfiguration config)  {  \_config = config;  }  // Here we use Generics so it picks up whatever data type is passed.  // In this case, the data type if the Computer class itself.  public T LoadDataSingle<T>(string sqlSelect)  {  IDbConnection dbConnection = new SqlConnection(\_config.GetConnectionString("DefaultConnection"));  return dbConnection.QuerySingle<T>(sqlSelect);  }  public IEnumerable<T> LoadData<T>(string sqlSelect)  {  IDbConnection dbConnection = new SqlConnection(\_config.GetConnectionString("DefaultConnection"));  return dbConnection.Query<T>(sqlSelect);  }  public bool InsertRow(string sqlString)  {  IDbConnection dbConnection = new SqlConnection(\_config.GetConnectionString("DefaultConnection"));  return dbConnection.Execute(sqlString) > 0;  }  }  } |

* + - 1. Here is the updated DataContextEF.cs file. Again the changes will be in blue:

|  |
| --- |
| using Computers.Models;  using Microsoft.EntityFrameworkCore;  using Microsoft.Extensions.Configuration;  namespace Computers.Data  {  public class DataContextEF : DbContext  {  private IConfiguration \_config;  public DataContextEF(IConfiguration config)  {  \_config = config;  }  public DbSet<Computer>? computer { get; set; }  protected override void OnConfiguring(DbContextOptionsBuilder options)  {  // When the DataContextEF class is created, we check to see if it has been configured. if not, we configure it.  if (!options.IsConfigured)  {  options.UseSqlServer(\_config.GetConnectionString("DefaultConnection"), options => options.EnableRetryOnFailure());  }  }  // ModelBuilder maps an actual model to a table  protected override void OnModelCreating(ModelBuilder modelBuilder)  {  // Set our Schema name  modelBuilder.HasDefaultSchema("TutorialAppSchema");  // Map the entity directly to the 'Computer' table within the 'TutorialAppSchema' schema.  modelBuilder.Entity<Computer>()  .HasKey(c => c.Id);    /\*  Note: If you are using a table in another schema other than the one set as default,  you will need to use this code instead:  modelBuilder.Entity<Computer>().ToTable("TableName","SchemaName");  \*/  }  }  } |

* + - 1. When we run this program, we get an error message stating the program cannot find the appsettings.json file.
      2. Here is how to fix the issue:
         1. The computer is looking for this file in the bin/Debug/net9.0 directory, which is underneath the Computer directory where our program lives.
         2. Actually, this is where the appsettings.json file is supposed to be, but we put it in the Computer directory as we consider it part of our code.
         3. We need to tell the computer that each time we run this program, this file needs to be copied down to the net9.0 directory.
         4. To do that, we need to update the Computers.csproj file and make a change to the ItemGroup section.
         5. In the code below, I have put the additional lines that need to be copied to the ItemGroup portion of our csproj file:

|  |
| --- |
| <ItemGroup>  <None Update="appsettings.json">  <CopyToOutputDirectory>Always</CopyToOutputDirectory>  </None>  <PackageReference Include="Dapper" Version="2.1.66" />  <PackageReference Include="Microsoft.Data.SqlClient" Version="6.1.0" />  <PackageReference Include="Microsoft.EntityFrameworkCore" Version="9.0.7" />  <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="9.0.7" />  <PackageReference Include="Microsoft.Extensions.Configuration" Version="9.0.7" />  <PackageReference Include="Microsoft.Extensions.Configuration.Json" Version="9.0.7" />  </ItemGroup> |

* + - * 1. Now each time the program runs, a fresh copy of the appsettings.json file we be copied to the net9.0 directory before executing.

1. [Intermediate Part II](#TS04)

* 1. [Files](#TS04)
     1. In this section we will be discussing:
        1. Reading/Writing to files
           1. Just basic text
        2. Parsing JSON
           1. Deserialization
           2. Serialization
        3. Simple Model Mapping
           1. Automapper
     2. Reading/Writing to files
        1. There is a static class called File.
        2. Since it is static, we can access its methods directly.
        3. Some of these methods are:
           1. AppendAllLines
           2. Copy
           3. Create
           4. Exists
           5. Move
           6. Open
           7. ReadAllText
           8. WriteAllText
        4. There is another class called StreamWriter that we can configure to write to files and set the append value to True or False.
     3. Writing to a file
        1. Here is an example of how to write to a file:

|  |
| --- |
| string fileName = "C:/Udemy/C#/Dominic Tripodi/My Applications/Intermediate/Files/log.txt";  using StreamWriter openFile = new(fileName, append: true);  openFile.WriteLine(sqlString);  openFile.close(); |

* + 1. Reading a file  
       1. Likewise, we can use Streamreader to read from a file. Here are three examples of using StreamReader:

|  |
| --- |
| string fileName = "C:/Udemy/C#/Dominic Tripodi/My Applications/Intermediate/Files/log.txt";  Example 1: Reading a file line by line.  using (StreamReader reader = new StreamReader(fileName))  {  string line;  while ((line = reader.ReadLine()) != null)  {  Console.WriteLine(line);  }  }  }  }  Example 2: Reading the Entire File at Once  using System;  using System.IO;  class Program  {  static void Main()  {  using (StreamReader reader = new StreamReader(fileName))  {  string content = reader.ReadToEnd();  Console.WriteLine(content);  }  }  }  Example 3: Reading with Custom Encoding  using System;  using System.IO;  using System.Text;  class Program  {  static void Main()  {  using (StreamReader reader = new StreamReader(fileName, Encoding.UTF8))  {  string content = reader.ReadToEnd();  Console.WriteLine(content);  }  }  } |

* 1. [JSON](#TS04)
     1. Deserialize
        1. From text to a model
     2. Serialize
        1. From a model to text
     3. Newtonsoft.Json.Serialization
        1. We will discuss these options
     4. Examples of working with Json data
        1. Since we are working in a new namespace, we need to install our six packages.
           1. Close down VSC, install packages, and then relaunch VSC with code .
           2. dotnet add package Dapper
           3. dotnet add package Microsoft.data.sqlclient
           4. dotnet add package Microsoft.entityframeworkcore
           5. dotnet add package Microsoft.entityframeworkcore.sqlserver
           6. dotnet add package Microsoft.Extensions.Configuration
           7. dotnet add package Microsoft.Extensions.Configuration.Json
     5. There are two methods of pulling in JSON data:
        1. Using the Newtonsoft.Json.Serialization package. (JsonConvert)
        2. Using the System.text package (System.Text.Json.JsonSerializer)
        3. One thing of note: The Newtonsoft package does need to be installed before we start using it. To accomplish this, follow these steps:
           1. Close out our Visual Studio Code window.
           2. Then in our PowerShell console, enter this command:

dotnet add package Newtonsoft.Json

* + - * 1. Now restart VSC with Code .
      1. In the script below, we will be doing both serialization (Sending object data to a file) and deserialization (Loading Json file data into an object).

Example of Serializing and Deserializing Data Using the System.Text and Newtonsoft methods

|  |
| --- |
| using System.Text.Json;  using Files.Data;  using Microsoft.Extensions.Configuration;  using Newtonsoft.Json;  using Newtonsoft.Json.Serialization;  namespace Files  {  internal class Program  {  public static void Main(string[] args)  {  IConfiguration config = new ConfigurationBuilder().AddJsonFile("appsettings.json").Build();  DataContextDapper dapper = new DataContextDapper(config);  Computer myComputer = new Computer()  {  Id = 24,  Motherboard = "Z690",  HasWifi = true,  HasLTE = false,  ReleaseDate = DateTime.Now,  Price = 943.87m,  VideoCard = "RTX 2060"  };  string fileNameRead = "C:/Udemy/C#/Dominic Tripodi/JSON files/Computers.json";  string computerCopyNewtonsoftFile = "C:/Udemy/C#/Dominic Tripodi/JSON files/ComputersNewtonsoft.json";  string computerCopySystemFile = "C:/Udemy/C#/Dominic Tripodi/JSON files/ComputersSystem.json";  string computersJson = File.ReadAllText(fileNameRead);  // Deserializing: Loading Json file information into INumerable objects  // --------------------------------------------------------------------  // Deserializing with Newtonsoft  // Note: No camel case conversion is necessary with Newtonsoft as this method is built into JsonConvert.  IEnumerable<Computer>? computersNewtonsoft = JsonConvert.DeserializeObject<IEnumerable<Computer>>(computersJson);  // Deserializing with System.  // Here we need JsonSerializerOptions to convert the lower-case Json file keys to match the Pascal case object property names.  JsonSerializerOptions option = new JsonSerializerOptions();  option.PropertyNamingPolicy = JsonNamingPolicy.CamelCase;  IEnumerable<Computer>? computersSystem = System.Text.Json.JsonSerializer.Deserialize<IEnumerable<Computer>>(computersJson,option);  // ====================================================================================================================================  // Serializing: Copying the IEnumerable objects into a Json file  // -------------------------------------------------------------  // Note: Both these methods below ensure that the output Json files have the fields declared in Camel Case format.  // Serializing with NewtonSoft  JsonSerializerSettings settings = new JsonSerializerSettings();  settings.ContractResolver = new CamelCasePropertyNamesContractResolver();  string computerCopyNewtonsoft = JsonConvert.SerializeObject(computersNewtonsoft,settings);  File.WriteAllText(computerCopyNewtonsoftFile, computerCopyNewtonsoft);  // Serializing with System  JsonSerializerOptions options = new JsonSerializerOptions();  options.PropertyNamingPolicy = JsonNamingPolicy.CamelCase;  string computerCopySystem = System.Text.Json.JsonSerializer.Serialize(computersSystem,options);  File.WriteAllText(computerCopySystemFile, computerCopySystem);  }  }  } |

* + 1. Writing our Json data to SQL.
       1. In the script below, we cover will execute different things:
          1. Read the ComputersSnake SQL table and write each line out to the Computer SQL table.
          2. Read the ComputersSnake SQL table and create a new Computers.Json file.

Performing SQL table row Insertion and Json File Creation

|  |
| --- |
| using FilesToSQL.Data;  using FilesToSQL.Model;  using Microsoft.Extensions.Configuration;  using Newtonsoft.Json;  using Newtonsoft.Json.Serialization;  namespace FilesToSQL  {  internal class Program  {  public static void Main(string[] args)  {  IConfiguration config = new ConfigurationBuilder().AddJsonFile("appsettings.json").Build();  DataContextDapper dapper = new DataContextDapper(config);  string computerSnakeFile = "C:/Udemy/C#/Dominic Tripodi/JSON files/ComputersSnake.json";  string computersSnake = File.ReadAllText(computerSnakeFile);  // Writing data to SQL and Json file  string ComputersFile = "C:/Udemy/C#/Dominic Tripodi/JSON files/Computers.json";  string sqlread1 = @"select c.ComputerId as Id,c.Motherboard as Motherboard,c.HasWifi as HasWifi,c.HasLTE as HasLTE,c.ReleaseDate as ReleaseDate,  cs.Price as Price,c.VideoCard as VideoCard from TutorialAppSchema.Computers c  inner join TutorialAppSchema.ComputersSnake cs on c.Motherboard = cs.Motherboard  where c.motherboard = cs.motherboard  group by c.ComputerId,c.Motherboard,c.HasWifi,c.HasLTE,c.ReleaseDate,cs.Price,c.VideoCard  order by c.ComputerId";  string sqlread2 = "select ComputerId,Motherboard,HasWifi,HasLTE,ReleaseDate,Price,VideoCard from TutorialAppSchema.ComputersSnake";  string truncateComputer = "truncate table TutorialAppSchema.Computer";  // Load the Computer table  dapper.ExecuteNonQuery(truncateComputer);  IEnumerable<Computer>? computerData1 = dapper.LoadData<Computer>(sqlread1);  int newRowId = 0;  foreach (Computer computer in computerData1)  {  newRowId++;  string sql = @"INSERT INTO TutorialAppSchema.Computer (ComputerId,Motherboard,HasWifi,HasLTE,ReleaseDate,Price,VideoCard) values('"  + newRowId + "','" + computer.Motherboard + "','" + computer.HasWifi + "','" + computer.HasLTE + "','"  + computer.ReleaseDate + "','" + computer.Price + "','" + computer.VideoCard.Replace("'","''") + "')";  dapper.InsertRow(sql);  }  // Recreate the Json file in correct format  IEnumerable<Computer>? computerData2 = dapper.LoadData<Computer>(sqlread2);  foreach (Computer computer in computerData2)  {  JsonSerializerSettings settings = new JsonSerializerSettings();  settings.ContractResolver = new CamelCasePropertyNamesContractResolver();  string computerCopyNewtonsoft = JsonConvert.SerializeObject(computerData2, settings);  File.WriteAllText(ComputersFile, computerCopyNewtonsoft);  string text = File.ReadAllText(ComputersFile);  text = text.Replace("},{", "},\n{");  File.WriteAllText(ComputersFile, text);  }  }  }  } |

* 1. [Model Mapping](#TS04)
     1. First we need to add a new package for AutoMapper.
     2. Enter this command in a PowerShell windows opened to your project location:
        1. dotnet add package AutoMapper
     3. Understanding why we may need AutoMapper.
        1. In the previous section, when working with Json files, our Json key fields were spelled the same, but the casing was off.
        2. There may be times when you are dealing with another class where the fields are basically the same, but they are spelled differently.
           1. For instance, perhaps the CPUCores property in the other class is spelled cpu\_cores.
        3. In this case, our Json Camel Case correction methods just won’t work because the issue is deeper as the actually field names are spelt differently.
        4. In the diagram below, we have two Classes, Computers and ComputersSnake. Observe how the field names are different from each other.

Computers.cs ComputersSnake.cs

|  |  |
| --- | --- |
| public class Computer  {  public int ComputerId { get; set; }  public string? Motherboard { get; set; } = "";  public int CPUCores { get; set; }  public bool HasWifi { get; set; }  public bool HasLTE { get; set; }  public DateTime? ReleaseDate { get; set; }  public decimal Price { get; set; }  public string? VideoCard { get; set; } = "";  } | public class ComputerSnake  {  public int computer\_id { get; set; }  public string? motherboard { get; set; } = "";  public int cpu\_cores { get; set; }  public bool has\_wifi { get; set; }  public bool has\_lte { get; set; }  public DateTime? release\_date { get; set; }  public decimal price { get; set; }  public string? video\_card { get; set; } = "";  } |

* + - 1. As you can see, we will have an issue referring each field name in Computers.cs to its peer name in ComputersSnake.cs.
    1. Overview of AutoMapper components:
       1. MapperConfiguration
       2. Mapper
       3. LoggerFactory
    2. Breakdown of each component:
       1. MapperConfiguration
          1. This has two constructor parameters:

Configuration

Normally denoted by the name cfg.

Configuration has a CreateMap() method which takes two parameters, those being the Source and Destination classes.

Here is an example of a configuration line where we will be mapping between the ComputerSnake and Computer classes:

cfg.CreateMap<ComputerSnake, Computer>()

Logging

Recently, logging has been introduced into MapperConfiguration.

Creating the logging is a two part process:

Create the ILoggerFactory object:

ILoggerFactory factory = new LoggerFactory();

Use the object as the second parameter of MapperConfiguration:

factory = LoggerFactory.Create(builder => builder.AddConsole())

* + - * 1. Configuration breakdown:

The configuration creates a map between the members (These are the two classes we are mapping together).

This mapping method is called ForMember().

ForMember takes a lambda expression takes two parameters, those being destination and option.

There are as many ForMember() methods as there are parameters in the two fields that will be mapped.

Here is an example of a ForMember statement

ForMember(destination => destination.ComputerId, options => options.MapFrom(source => source.computer\_id))

Here, the destination ComputerId field maps from the source computer\_id field.

Changes can also be made during the mapping process as well. For example if we map the price field, we can make the destination price 80 percent of the source:

ForMember(destination => destination.Price, options => options.MapFrom(source => source.price \* 0.8))

* + 1. Example of using mapping
       1. In our example below, we will read in a Json file which has the field names matching those in the ComputerSnake class listed above.
       2. Our objective is to map them to the Computer class object using Newtonsoft deserialization.
       3. Once the Json lines have been turned into objects, we will be saving them to an SQL table called Computers.

Using AutoMapper to Map ComputerSnake to Computer and Write Data to SQL

|  |
| --- |
| using AutoMapper;  using FilesToSQL.Data;  using FilesToSQL.Model;  using Microsoft.Extensions.Configuration;  using Microsoft.Extensions.Logging;  using Newtonsoft.Json;  namespace FilesToSQL  {  internal class Program  {  public static void Main(string[] args)  {  IConfiguration config = new ConfigurationBuilder().AddJsonFile("appsettings.json").Build();  DataContextDapper dapper = new DataContextDapper(config);  string computersSnake = "C:/Udemy/C#/Dominic Tripodi/JSON files/ComputersSnake.json";  string computersJson = File.ReadAllText(computersSnake);  ILoggerFactory factory = new LoggerFactory();  Mapper mapper = new Mapper(new MapperConfiguration((cfg) =>  {  cfg.CreateMap<ComputerSnake, Computer>()  .ForMember(destination => destination.ComputerId, options =>  options.MapFrom(source => source.computer\_id))  .ForMember(destination => destination.Motherboard, options =>  options.MapFrom(source => source.motherboard))  .ForMember(destination => destination.CPUCores, options =>  options.MapFrom(source => source.cpu\_cores))  .ForMember(destination => destination.HasWifi, options =>  options.MapFrom(source => source.has\_wifi))  .ForMember(destination => destination.HasLTE, options =>  options.MapFrom(source => source.has\_lte))  .ForMember(destination => destination.ReleaseDate, options =>  options.MapFrom(source => source.release\_date))  .ForMember(destination => destination.Price, options =>  options.MapFrom(source => source.price))  .ForMember(destination => destination.VideoCard, options =>  options.MapFrom(source => source.video\_card));  }, factory = LoggerFactory.Create(builder => builder.AddConsole())));  // Recreate the Json file in correct format  // System.Text Deserializer  // IEnumerable<ComputerSnake>? computersSystem = System.Text.Json.JsonSerializer.Deserialize<IEnumerable<ComputerSnake>>(computersJson);  //Newtonsoft Deserializer  IEnumerable<ComputerSnake>? computersSystem = JsonConvert.DeserializeObject<IEnumerable<ComputerSnake>>(computersJson);  if (computersSystem != null)  {  IEnumerable<Computer> computerResult = mapper.Map<IEnumerable<Computer>>(computersSystem);    // Clear out the Computers table  string truncateComputer = "truncate table TutorialAppSchema.Computer";  dapper.ExecuteNonQuery(truncateComputer);  foreach (Computer computer in computerResult)  {  if (computer.VideoCard == null) { computer.VideoCard = "";}  if (computer.Motherboard == null) { computer.Motherboard = ""; }  string sql = @"INSERT INTO TutorialAppSchema.Computer (ComputerId,Motherboard,HasWifi,HasLTE,ReleaseDate,Price,VideoCard) values('"  + computer.ComputerId + "','" + computer.Motherboard + "','" + computer.HasWifi + "','" + computer.HasLTE + "','"  + computer.ReleaseDate + "','" + computer.Price + "','" + computer.VideoCard.Replace("'", "''") + "')";  dapper.InsertRow(sql);  }  }  }  }  } |

* + 1. Using Json properties to map fields.
       1. We can omit AutoMapper altogether by using Json properties.
       2. Notice the diagram below. In our Computer class above each property, we have a JsonPropertyName tag that maps that converts incoming property names which have that label.

Example of Mapping using Json Properties

|  |  |
| --- | --- |
| public class Computer  {  [JsonPropertyName(“computer\_id”)]  public int ComputerId { get; set; }  [JsonPropertyName(“motherboard”)]  public string? Motherboard { get; set; } = "";  [JsonPropertyName(“cpu\_cores”)]  public int CPUCores { get; set; }  [JsonPropertyName(“has\_wifi”)]  public bool HasWifi { get; set; }  [JsonPropertyName(“has\_lte”)]  public bool HasLTE { get; set; }  [JsonPropertyName(“release\_date”)]  public DateTime? ReleaseDate { get; set; }  [JsonPropertyName(“price”)]  public decimal Price { get; set; }  [JsonPropertyName(“video\_card”)]  public string? VideoCard { get; set; } = "";  } | public class ComputerSnake  {  public int computer\_id { get; set; }  public string? motherboard { get; set; } = "";  public int cpu\_cores { get; set; }  public bool has\_wifi { get; set; }  public bool has\_lte { get; set; }  public DateTime? release\_date { get; set; }  public decimal price { get; set; }  public string? video\_card { get; set; } = "";  } |

* 1. [Async and Await](#TS04)
     1. These are methods of starting tasks asynchronously in a separate thread.
     2. For an asynchronous task to be created, the method it is in has to be defined as async.
     3. A task must be created and started.
     4. Finally, an await line must be created to ensure the program does not end before the async task has completed.
     5. Below is a simple program showing how async works.

Example of Using Async and Await

|  |
| --- |
| using System;  namespace FilesToSQL  {  internal class Program  {  public static async Task Main(string[] args)  {  Task firstTask = new Task(() =>  {  Thread.Sleep(100);  Console.WriteLine("Task 1");  });  firstTask.Start();  Console.WriteLine("After the Task was created");  await firstTask;  }  }  } |

Example of using Async and ConsoleAfterDelayAsync

|  |
| --- |
| using System;  namespace FilesToSQL  {  internal class Program  {  public static async Task Main(string[] args)  {  Task firstTask = new Task(() =>  {  Thread.Sleep(100);  Console.WriteLine("Task 1");  });  firstTask.Start();  Task secondTask = ConsoleAfterDelayAsync("Task 2", 150);  ConsoleAfterDelay("Delay", 101);  Task thirdTask = ConsoleAfterDelayAsync("Task 3", 50);  await secondTask;  await firstTask;  Console.WriteLine("After the Task was created");  await thirdTask;  }  static void ConsoleAfterDelay(string text, int delayTime)  {  Thread.Sleep(delayTime);  Console.WriteLine(text);  }  static async Task ConsoleAfterDelayAsync(string text, int delayTime)  {  Thread.Sleep(delayTime);  Console.WriteLine(text);  }  }  } |

1. [SQL Intermediate](#TS05)

* 1. [Backing up and restoring a database](#TS05)
     1. Backing up the DotNetCourseDatabase to disk:
        1. BACKUP DATABASE [DotNetCourseDatabase] TO DISK = 'C:\Udemy\C#\Dominic Tripodi\Database Backups\DotNetCourseDatabase.bak' WITH format,medianame = 'SQLServerBackups',name = 'Full DotNetCourseDatabaseBackup'
     2. Restoring the DotNetCourseDatabase from disk:
        1. USE MASTER
        2. GO
        3. RESTORE DATABASE [DotNetCourseDatabase] FROM DISK = N'C:\Udemy\C#\Dominic Tripodi\Database Backups\DotNetCourseDatabase.bak' WITH FILE = 1, NOUNLOAD, STATS = 5
        4. GO

* 1. [Joins](#TS05)
     1. This is a way of joining two tables together based on a common field.
     2. There are typically three types of join statements:
        1. Left
        2. Inner
        3. Right
     3. Often we have a master table and detailed table which both contain a common field.
     4. For example, a car tune-up shop may have a master table containing customers information.
     5. One of the fields in this master table is CustomerId.
     6. Another table holds all the works done for every single customer, detailing every type of work.
        1. This table has many columns (PartNo, HoursWorked … etc). One of these fields is also CustomerId
     7. To pull all the work performed for a particular customer, we would join the two tables together based on the CustomerId column.
        1. This would display the customers information from the master table as well as pulling the related customers information from the detailed table.
     8. We can have more than one join in a query as well.
     9. Below is a sample join statement using both Inner and Left join statements.
        1. This query will only show Salaried folks (Inner JOIN) but will display all job information (Left JOIN):

Example of Using Both Inner and Left Joins Statements

|  |
| --- |
| SELECT Users.UserId  , Users.FirstName + ' ' + Users.LastName AS FullName  , UserJobInfo.JobTitle  , ISNULL(UserJobInfo.Department,'No Dept Listed')  , UserSalary.Salary  , Users.Email  , Users.Gender  , Users.Active  FROM TutorialAppSchema.Users AS Users  INNER JOIN TutorialAppSchema.UserSalary AS UserSalary  ON UserSalary.UserId = Users.UserId  LEFT JOIN TutorialAppSchema.UserJobInfo AS UserJobInfo  ON UserJobInfo.UserId = Users.UserId  WHERE Users.Active = 1  ORDER BY Users.UserId DESC; |

* 1. [JOIN verses WHERE](#TS05)
     1. We can often get the same result using WHERE and we can with JOIN, and use less resources on the system in doing so.
     2. Here are two examples of pulling UserID and Salary from the UserSalary tables based related records in the UserJobInfo table which render the same results:

JOIN Command

|  |
| --- |
| SELECT us.UserId,us.Salary FROM TutorialAppSchema.UserSalary us  INNER JOIN TutorialAppSchema.UserJobInfo uj ON us.UserId = uj.UserId |

Where Command

|  |
| --- |
| SELECT [UserSalary].[UserId],[UserSalary].[Salary] FROM  TutorialAppSchema.UserSalary AS UserSalary WHERE EXISTS (  SELECT \* FROM TutorialAppSchema.UserJobInfo AS UserJobInfo  WHERE UserJobInfo.UserId = UserSalary.UserId) |

* 1. [Creating indexes](#TS05)
     1. We can speed up our queries by adding indexes on particular columns within a table.
     2. There are three types of indexes available. These are
        1. Clustered Index
           1. By default, the primary key of a table becomes a clustered index automatically.
           2. This will physically cause the table to be reordered based on that particular column.

It is for this reason that there can only be one clustered index within a table.

The naming convention for a clustered index is for the index name to begin with cix\_ which denotes it as clustered.

* + - * 1. Below we will create a clustered index named cix\_UserSalary\_UserId. It will be made on the UserID field within the UserSalary table:

CREATE CLUSTERED INDEX cix\_UserSalary\_UserId ON TutorialAppSchema.UserSalary(UserId)

* + - 1. Non-Clustered Index
         1. These are the default types of indexes.
         2. The naming convention for a non-clustered index is for the index name to begin with ix\_ which denotes it as non-clustered.
         3. Oddly enough, the field we create the clustered index on will also contain the Clustered index.
         4. Below we will create a non-clustered index on the JobTitle field within the UserJobInfo table.
         5. This index will also be storing the Department field within the index.

CREATE NONCLUSTERED INDEX ix\_UserJobInfo\_JobTitle ON TutorialAppSchema.UserJobInfo(JobTitle) INCLUDE (Department)

* + - 1. Filtered Index
         1. This is a non-clustered index which contains a WHERE clause.
         2. This makes the query run faster as the WHERE clause will filter out records that do not meet the search criteria of the WHERE clause.
         3. The naming convention for a filtered index is for the index name to begin with fix\_ which denotes it as filtered.
         4. Below is an example of a filtered index on the Active column of the Users table where Active = 1:

CREATE NONCLUSTERED INDEX fix\_Users\_Active ON Tutorial.Users(Active) INCLUDE ([Email], [FirstName], [LastName]) WHERE Active = 1

* 1. [Outer Apply and Cross Apply](#TS05)
     1. Outer Apply
        1. This clause in SQL Server allows you to perform a left join of a table with a table-valued function or a correlated subquery.
        2. It works similarly to a LEFT JOIN but is used when you need to join a table with a table-valued function or a subquery that depends on the outer query.
     2. Cross Apply
        1. This function in SQL Server is a powerful feature that allows you to join a table with a table-valued function or a correlated subquery.
        2. It is particularly useful when you need to evaluate a table-valued expression for each row of the left table expression.
     3. The following SQL statement demonstrates these two methods.

Demonstration of the Use of Outer Apply and Cross Apply

|  |
| --- |
| SELECT Users.UserId  , Users.FirstName + ' ' + Users.LastName AS FullName  , UserJobInfo.JobTitle  , UserJobInfo.Department  , DepartmentAverage.AvgSalary  , UserSalary.Salary  , Users.Email  , Users.Gender  , Users.Active  FROM TutorialAppSchema.Users AS Users  --INNER JOIN  JOIN TutorialAppSchema.UserSalary AS UserSalary  ON UserSalary.UserId = Users.UserId  LEFT JOIN TutorialAppSchema.UserJobInfo AS UserJobInfo  ON UserJobInfo.UserId = Users.UserId  -- OUTER APPLY ( -- Similar to LEFT JOIN  CROSS APPLY ( -- Similar to JOIN  -- SELECT TOP 1  SELECT ISNULL (UserJobInfo2.Department, 'No Department Listed') AS Department  , AVG (UserSalary2.Salary) AS AvgSalary  FROM TutorialAppSchema.UserSalary AS UserSalary2  LEFT JOIN TutorialAppSchema.UserJobInfo AS UserJobInfo2  ON UserJobInfo2.UserId = UserSalary2.UserId  WHERE UserJobInfo2.Department = UserJobInfo.Department  GROUP BY UserJobInfo2.Department  ) AS DepartmentAverage  WHERE Users.Active = 1  ORDER BY Users.UserId DESC; |

* 1. [Date Examples](#TS05)
     1. SELECT GETDATE()
        1. Returns the current date where the SQL server is living.
     2. SELECT DATEADD(YEAR, -5, GETDATE())
        1. Returns the date exactly 5 years ago
     3. SELECT DATEDIFF(MINUTE, DATEADD(YEAR, -5, GETDATE()), GETDATE())
        1. Returns the number of minutes that have elapsed between exactly five years ago and now.

* 1. [Alter Table](#TS05)
     1. This is a means of changing the structure of the table itself.
     2. In the example below, we will alter the UserSalary table by adding the AvgSalary field:
        1. ALTER TABLE TutorialAppSchema.UserSalary ADD AvgSalary DECIMAL(18, 4);
     3. Now we will populate the AvgSalary cells with some code that determines average salary for a department

Example of Populating the AvgSalary Fields with Average Salay for a Department

|  |
| --- |
| UPDATE UserSalary  SET UserSalary.AvgSalary = DepartmentAverage.AvgSalary  FROM TutorialAppSchema.UserSalary AS UserSalary  LEFT JOIN TutorialAppSchema.UserJobInfo AS UserJobInfo  ON UserJobInfo.UserId = UserSalary.UserId  CROSS APPLY ( -- Similar to JOIN  -- SELECT TOP 1  SELECT ISNULL (UserJobInfo2.Department, 'No Department Listed') AS Department  , AVG (UserSalary2.Salary) AS AvgSalary  FROM TutorialAppSchema.UserSalary AS UserSalary2  LEFT JOIN TutorialAppSchema.UserJobInfo AS UserJobInfo2  ON UserJobInfo2.UserId = UserSalary2.UserId  WHERE ISNULL (UserJobInfo2.Department, 'No Department Listed') = ISNULL (UserJobInfo.Department, 'No Department Listed')  GROUP BY UserJobInfo2.Department  ) AS DepartmentAverage; |

1. [API Basics](#TS06)

* 1. [Introduction](#TS06)
     1. First, create and navigate to the following path in PowerShell:
        1. C:\Udemy\C#\Dominic Tripodi\My Applications\Basic API
     2. Type theses command in the PowerShell console:
        1. dotnet new webapi -n DotnetAPI
        2. cd DotnetAPI
        3. code .
     3. Your Visual Studio Code screen will now open.
     4. Open the Program.cs file in VSC.
     5. Let’s review the lines in Program.cs one by one.
        1. var builder = WebApplication.CreateBuilder(args);
           1. Builds the server where the API lives
        2. builder.Services.AddOpenApi();
           1. We will add Swagger here.
        3. if (app.Environment.IsDevelopment()) { }
           1. Build additional application here
        4. app.UseHttpsRedirection();
           1. When the application goes to production, we put it behind https.
        5. The remaining lines up to app.Run(); will be removed as we will be implementing Swagger.
        6. App.Run();
           1. This launches the application.
        7. record WeatherForecast(DateOnly Date, int TemperatureC, string? Summary)
           1. This is just our example weather object offered by the initialization of the Program.cs file

* 1. [Installing Swagger](#TS06)
     1. Close the VSC window
     2. Back at the PowerShell screen, enter these commands:
        1. dotnet add package Swashbuckle.AspNetCore
        2. code .
     3. Now that our VSC application is up, open the DotnetAPI.csproj file and review the ItemGroup section .
     4. You should see this line:

|  |
| --- |
| <ItemGroup>  <PackageReference Include="Microsoft.AspNetCore.OpenApi" Version="9.0.7" />  <PackageReference Include="Swashbuckle.AspNetCore" Version="9.0.3" />  </ItemGroup> |

1. Back in Program.cs, enter this line underneath the var builder = WebApplication.CreateBuilder(args); line
   * + 1. builder.Services.AddEndpointsApiExplorer();
       2. builder.Services.AddSwaggerGen();
     1. Inside the app.Environment.IsDevelopment()) brackets, remove the app.MapOpenApi(); line and enter these two lines:
        1. app.UseSwagger();
        2. app.UseSwaggerUI();

* 1. [Starting Swagger and creating a controller](#TS06)
     1. At the PowerShell window, enter dotnet run
     2. The program will run … and run … and run.
     3. The point is, it stays up until we shut it down because this is the listener code for Swagger.
     4. On the PowerShell screen, you will see this:

|  |
| --- |
| info: Microsoft.Hosting.Lifetime[14]  Now listening on: http://localhost:5263  info: Microsoft.Hosting.Lifetime[0]  Application started. Press Ctrl+C to shut down.  info: Microsoft.Hosting.Lifetime[0]  Hosting environment: Development  info: Microsoft.Hosting.Lifetime[0]  Content root path: C:\Udemy\C#\Dominic Tripodi\My Applications\Basic API\DotnetAPI |

* + 1. Notice the text is blue: <http://localhost:5263>
       1. This is the listening address on your computer.
       2. Currently it is running on port 5263.
       3. Each time you run the program, you might get a different port number, which is fine.
    2. Now, open a browser window and enter the address that is on the screen. In this case it is <http://localhost:5263>
       1. This is the screen you will see:

|  |
| --- |
|  |

* + - 1. This is fine as we haven’t set anything up yet for the link to point to.
      2. We can however access the built in Weather Forecast object with Program.cs which looks like this:

|  |
| --- |
| record WeatherForecast(DateOnly Date, int TemperatureC, string? Summary)  {  public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);  } |

* + - 1. To get to it, we just add WeatherForecast to the end of our http line like so: <http://localhost:5263/WeatherForecast>
      2. We will get back some Json text relating to the weather forecast which looks like this:

|  |
| --- |
| [{"date":"2025-07-30","temperatureC":-8,"summary":"Sweltering","temperatureF":18},  {"date":"2025-07-31","temperatureC":21,"summary":"Bracing","temperatureF":69},  {"date":"2025-08-01","temperatureC":45,"summary":"Chilly","temperatureF":112},  {"date":"2025-08-02","temperatureC":20,"summary":"Warm","temperatureF":67},  {"date":"2025-08-03","temperatureC":25,"summary":"Mild","temperatureF":76}] |

* + - 1. As titillating as this is, what we want to see is the Swagger page. We get there by replacing WeatherForecast with Swagger
         1. The full link is: <http://localhost:port#/swagger/index.html>
      2. The browser page will now look like this:

|  |
| --- |
|  |

* 1. [Setting up a controller](#TS06)
     1. In VSC, create a new folder and call it Controllers
     2. In that new folder, create a new file called WeatherForecastController.cs
     3. So what we want to do (and this will be only for testing purposes) is to build a new controller to hold our Weather Forecast.
     4. We will be piecemealing the original weather forecast code from the Program.cs file and placing it in our new WeatherForecastController.cs file.
     5. From our Program.cs file, the lines in blue we will be removing and putting in our controller file:

Lines to Remove From the Program.cs File

|  |
| --- |
| var builder = WebApplication.CreateBuilder(args);  // Add services to the container.  // Learn more about configuring OpenAPI at https://aka.ms/aspnet/openapi  builder.Services.AddOpenApi();  builder.Services.AddEndpointsApiExplorer();  builder.Services.AddSwaggerGen();  var app = builder.Build();  // Configure the HTTP request pipeline.  if (app.Environment.IsDevelopment())  {  app.UseSwagger();  app.UseSwaggerUI();  }  app.UseHttpsRedirection();  var summaries = new[]  {  "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot", "Sweltering", "Scorching"  };  app.MapGet("/weatherforecast", () =>  {  var forecast = Enumerable.Range(1, 5).Select(index =>  new WeatherForecast  (  DateOnly.FromDateTime(DateTime.Now.AddDays(index)),  Random.Shared.Next(-20, 55),  summaries[Random.Shared.Next(summaries.Length)]  ))  .ToArray();  return forecast;  })  .WithName("GetWeatherForecast");  app.Run();  record WeatherForecast(DateOnly Date, int TemperatureC, string? Summary)  {  public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);  } |

* + 1. This is the code that will build out our WeatherForecastController.cs file to make it render the Weather Forecast page:

Updated WeatherForecastController.cs file

|  |
| --- |
| using Microsoft.AspNetCore.Mvc;  namespace DotnetAPI.Controllers;  [ApiController]  [Route("[controller]")]  public class WeatherForecastController : ControllerBase  {  private readonly string[] \_summaries = new[]  {  "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot", "Sweltering", "Scorching"  };  [HttpGet("", Name = "GetWeatherForecast")]  public IEnumerable<WeatherForecast> GetFiveDayForecast()  {  var forecast = Enumerable.Range(1, 5).Select(index =>  new WeatherForecast  (  DateOnly.FromDateTime(DateTime.Now.AddDays(index)),  Random.Shared.Next(-20, 55),  \_summaries[Random.Shared.Next(\_summaries.Length)]  ))  .ToArray();  return forecast;  }  }  public record WeatherForecast(DateOnly Date, int TemperatureC, string? Summary)  {  public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);  } |

* + 1. Now we need to update our Program.cs file to do two things:
       1. Build the AddControllers service
       2. Add the MapControllers method to app.
    2. The newly updated Program.cs file is shown below with the two new services listed above in blue:

Program.cs File with AddController Service and Map Controllers Method added

|  |
| --- |
| var builder = WebApplication.CreateBuilder(args);  builder.Services.AddControllers();  // Add services to the container.  // Learn more about configuring OpenAPI at https://aka.ms/aspnet/openapi  builder.Services.AddOpenApi();  builder.Services.AddEndpointsApiExplorer();  builder.Services.AddSwaggerGen();  var app = builder.Build();  // Configure the HTTP request pipeline.  if (app.Environment.IsDevelopment())  {  app.UseSwagger();  app.UseSwaggerUI();  }  app.MapControllers();  app.Run(); |

* + 1. Now we stop the PowerShell program with Ctrl/C and run it again with dotnet run.

* 1. [Creating our First Custom Controller](#TS06)
     1. In the Controllers directory, create a new file called UserController.cs.
     2. This is what it should look like:

Example of Initial UserController.cs File

|  |
| --- |
| using Microsoft.AspNetCore.Mvc;  namespace DotnetAPI.Controllers;  [ApiController]  [Route("[controller]")]  public class UserController : ControllerBase  {  UserController()  {  }  [HttpGet("test")]  public string[] Test()  {  string[] responseArray = new string[] {  "test1","test2"  };  return responseArray;  }  } |

* + 1. One thing to consider here is how the UserController.cs file differs from the WeatherForecastController.cs file.
       1. Our new UserController.cs file *has a method*.
          1. This method Test is denoted by the [HttpGet("test")] tag above it.
       2. This is important to note as methods are reachable by a backslash after the initial controller name.

* 1. [Shutdown and restart the application](#TS06)
     1. This can be done using Ctrl/C and dotnet run.
     2. If we go to our swagger link in our browser and refresh, you will see what I mean.
     3. In the diagram below, the user controller has a /test link in it.

|  |
| --- |
|  |

* + 1. If we click on the Try It Out button, we get the following error:
       1. System.InvalidOperationException: A suitable constructor for type 'DotnetAPI.Controllers.UserController' could not be located.
    2. Why did we get this error?
       1. It is because we did not make the constructor public.
       2. Remember to **always** make the constructor public in your Controller files!
    3. Make the constructor public and Ctrl/C the application.
    4. When we restart our application this time, we will be using a different command than dotnet run.
       1. Enter dotnet *watch* run instead.
       2. Notice our console screen now as shown below:  
          A screen shot of a computer

          AI-generated content may be incorrect.
       3. The Hot reload enabled information message is displayed.
       4. This means when we make a change we do not have to restart the program as they will automatically be reload.
          1. Note: You can force a manual reload by typing Ctrl/R in the console window.
    5. Back in our UserController.cs file, update the method declaration line as shown below:
       1. Old line: public string[] Test()
       2. New line: public string[] Test(string testValue)
    6. We now have added input parameters which means the user can pass something through the browser URL window.
    7. This means we can pass a value to our User link by the user of a question mark:
       1. http://localhost:5263/user/test?testValue="Hello World"
    8. This does not render a very friendly front end page for the user.
    9. We are now going to make another change to UserController.cs, this time to the HttpGet line:
       1. Old line: [HttpGet("test")]
       2. New line: [HttpGet("test/{testValue}")]
    10. We will also make a change to our text line in the code by updating out text1, text2 line to this: "test1","test2",testValue
    11. Now if we refresh our swagger browser window.
    12. In swagger, we can go to our User page, select try it out, then click on execute.
        1. You will see a text box for you to enter your own text.
        2. When you execute the page, whatever you entered will be displayed in the results window!
    13. Finally in our UserController.cs file, we change the HttpGet parameter and method name from test to GetUsers to make it more applicable to its purpose.
    14. This is what our UserController.cs file should look like now:

Updated Version of the UserController.cs File

|  |
| --- |
| using Microsoft.AspNetCore.Mvc;  namespace DotnetAPI.Controllers;  [ApiController]  [Route("[controller]")]  public class UserController : ControllerBase  {  public UserController()  {  }  [HttpGet("test/{testValue}")]  public string[] Test(string testValue)  {  string[] responseArray = new string[] {  "test1","test2",testValue  };  return responseArray;  }  } |

* + 1. The takeaway here is we can create our own customer controller and pass data to them.

* 1. [Updating the LaunchSettings.json file](#TS06)
     1. In VSC, open the launchSettings.json file which is in the properties folder.
     2. It will look like this:

The launchSettings.json File

|  |
| --- |
| {  "$schema": "https://json.schemastore.org/launchsettings.json",  "profiles": {  "http": {  "commandName": "Project",  "dotnetRunMessages": true,  "launchBrowser": false,  "applicationUrl": "http://localhost:5263",  "environmentVariables": {  "ASPNETCORE\_ENVIRONMENT": "Development"  }  },  "https": {  "commandName": "Project",  "dotnetRunMessages": true,  "launchBrowser": false,  "applicationUrl": "https://localhost:7150;http://localhost:5263",  "environmentVariables": {  "ASPNETCORE\_ENVIRONMENT": "Development"  }  }  }  } |

* + 1. What we want to do is force our application to run at port 5000 for http and port 5001 for https.
       1. In the http section, update the applicationUrl line so it looks like this:
          1. "applicationUrl": "http://localhost:5000",
       2. In the https section, change the applicationUrl line so it looks like this:
          1. "applicationUrl": "https://localhost:5001;http://localhost:500",
       3. If you want the swagger web page to load each time you make a code change, you could make this change in both http and https:
          1. "launchBrowser": true,
          2. “launchUrl”: “swagger”,
    2. Launching the application in HTTP and HTTPS
       1. Currently our application launches in http mode because of Swagger being added to the isDevelopment settings in Program.cs.
       2. We can explicitly choose whose to run in by entering this command when we launch the application in PowerShell:
          1. dotnet watch run –launch-profile http
          2. dotnet watch run –launch-profile https
       3. Again, by default the application launches in http mode when we enter the command: dotnet watch run
    3. The takeaway from this section is:
       1. We want to force our application to run on port 500 for http and 5001 for https:
       2. We can force launchSettings.json to automatically launch swagger when we start the application.
       3. We can override the environment that launches by adding the –launch-profile switch to dot watch run

1. [CORS](#TS07)

* 1. [CORS (or Cross Origin Resource Sharing) is a method that gives us access to a number of resources in our application](#TS07).
     1. It allows us to share our application with various front-end applications.
     2. To do this we will need to update our Program.cs file.
     3. In Program.cs change these lines:

|  |
| --- |
| var app = builder.Build();  if (app.Environment.IsDevelopment())  {  app.UseSwagger();  app.UseSwaggerUI();  }  else  {  app.UseHttpsRedirection();  } |

* + 1. To these lines:

|  |
| --- |
| builder.Services.AddCors((options) =>  {  options.AddPolicy("DevCors", (corsBuilder) =>  {  // Port 4200: Angular, Port 3000: Node.JS, Port 8000 : Django  corsBuilder.WithOrigins("http://localhost:4200", "http://localhost:3000", "http://localhost:8000")  .AllowAnyMethod()  .AllowAnyHeader()  .AllowCredentials();  });  options.AddPolicy("ProdCors", (corsBuilder) =>  {  corsBuilder.WithOrigins("https://myProductionSite.com")  .AllowAnyMethod()  .AllowAnyHeader()  .AllowCredentials();  });  });    var app = builder.Build();    // Configure the HTTP request pipeline.  if (app.Environment.IsDevelopment())  {  app.UseCors("DevCors");  app.UseSwagger();  app.UseSwaggerUI();  }  else  {  app.UseCors("ProdCors");  app.UseHttpsRedirection();  } |

* 1. Let’s talk a bit about these new lines.
     1. Building the AddCors service line
        1. builder.Services.AddCors((options) =>
        2. This line will receive the options which are compiled within the curly brackets.
        3. This line will take all the option settings and put them into the app method which builds the overall applications.
     2. AddPolicy line
        1. options.AddPolicy("DevCors", (corsBuilder) =>
        2. This is an anonymous function which takes parameters in both the dev and prod sections.
     3. WithOrigins line
        1. corsBuilder.WithOrigins("http://localhost:4200", "http://localhost:3000", "http://localhost:8000")
        2. These local host ports are commonly used front-end applications
           1. 4200 is the default port for Angular
           2. 3000 is the default port for React
           3. 8000 is the default port for View.
        3. This takes in three parameters:
           1. AllowAnyMethod

Refers to the different verbs we can send to our API.

These verbs include: GET, POST, PATCH, DELETE and so on.

* + - * 1. AllowAnyHeader

Allows user to pass any dynamic headers.

* + - * 1. AllowCredentials

Allows users to pass tokens.

* + 1. Adding our CORS policy to the app builder.
       1. Inside the if (app.Environment.IsDevelopment()) {} area, we see this line:
          1. app.UseCors("DevCors");
       2. This line takes everything bundled up in the DevCors policy and applies it to the app service.

1. [Database Connections](#TS08)
   1. We will go over basic database connectivity here, but please review the database section link [here](#S03_03) for more detailed database setup instructions. .
   2. Adding the connection string to appsettings.json
      1. If you remember if previous sections where we were dealing with database connectivity, we needed to add a construction to the appsettings.json file.
         1. It is the same here.
      2. Below is how our appsettings.json file will look after we added the connection string:

Example of Updating appsettings.json to Add the Database Connection String

|  |
| --- |
| {  "ConnectionStrings": {  "DefaultConnection":"Server=DAVES\_PC;Database=DotNetCourseDatabase;TrustServerCertificate=true;Trusted\_Connection=true;"  },  "Logging": {  "LogLevel": {  "Default": "Information",  "Microsoft.AspNetCore": "Warning"  }  },  "AllowedHosts": "\*"  } |

* 1. Next we need to install our packages needed for connectivity, so stop the application and add these packages into PowerShell.
     1. dotnet add package automapper --version 14
     2. dotnet add package dapper
     3. dotnet add package Microsoft.Data.SqlClient
     4. Restart the application with the dotnet watch run command.
  2. Next, bring up the UserController.ps file and make these changes.
     1. It should now look like this:

Structure of the UserController.cs File

|  |
| --- |
| using Microsoft.AspNetCore.Mvc;  namespace DotnetAPI.Controllers;  [ApiController]  [Route("[controller]")]  public class UserController : ControllerBase  {  DataContextDapper \_dapper;  // Without the builder statement in Program.cs file, we would need to issue this command to get the config.  // IConfiguration config = new ConfigurationBuilder().AddJsonFile("appSettings.json").Build();  // It is Program.cs and more specifically the var builder that does this automatically for us now.  // The constructor provides the private instance variable \_dapper the connection info from DataContextDapper.  public UserController(IConfiguration config)  {  // Here we supply \_dapper with the instantiation of DataContextDapper along with the connection string.  \_dapper = new DataContextDapper(config);  }  [HttpGet("TestConnection")]  public DateTime TestConnection()  {  return \_dapper.LoadDataSingle<DateTime>("Select DateTime()");  }    [HttpGet("test/{testValue}")]  public string[] Test(string testValue)  {  string[] responseArray = new string[] {  "test1","test2",testValue  };  return responseArray;  }  } |

* + 1. Next, Create a new Data folder.
    2. Create a new file named DataContextDapper,cs and place the following code in it:

Structure of the DataContextDapper.cs File

|  |
| --- |
| using System.Data;  using Dapper;  using Microsoft.Data.SqlClient;  namespace DotnetAPI  {  class DataContextDapper  {  private readonly IConfiguration \_config;  public DataContextDapper(IConfiguration config)  {  \_config = config;  }  // Here we use Generics so it picks up whatever data type is passed.  // In this case, the data type if the Computer class itself.  public T LoadDataSingle<T>(string sqlSelect)  {  IDbConnection dbConnection = new SqlConnection(\_config.GetConnectionString("DefaultConnection"));  return dbConnection.QuerySingle<T>(sqlSelect);  }  public IEnumerable<T> LoadData<T>(string sqlSelect)  {  IDbConnection dbConnection = new SqlConnection(\_config.GetConnectionString("DefaultConnection"));  return dbConnection.Query<T>(sqlSelect);  }  public bool InsertRow(string sqlString)  {  IDbConnection dbConnection = new SqlConnection(\_config.GetConnectionString("DefaultConnection"));  return dbConnection.Execute(sqlString) > 0;  }  }  } |

1. [User Models](#TS09)
   1. These models will mimic the tables within database as they are used as a template for ingesting incoming SQL data.
   2. These models will go inside of a folder called Model, so create that folder now.
   3. Since we have three database tables, we will be creating three Model classes:

Model to Database Table Cross Reference Chart

|  |  |
| --- | --- |
| Model Name | Associated Database Table |
| User | TutorialAppSchema.Users |
| UserSalary | TutorialAppSchema.UserSalary |
| UserJobInfo | TutorialAppSchema.UserJobInfo |

* 1. Now we will create our three Model classes:
     1. Users.cs
        1. The Users.cs Model class should be written as shown below

The User.cs File

|  |
| --- |
| namespace DotnetAPI  {  public partial class User  {  public int UserId { get; set; }  public string FirstName { get; set; }  public string LastName { get; set; }  public string EMail { get; set; }  public string Gender { get; set; }  public bool Active { get; set; }  public Users()  {  if (FirstName == null) { FirstName = ""; }  if (LastName == null) { LastName = ""; }  if (EMail == null) { EMail = ""; }  if (Gender == null) { Gender = ""; }  }  }  } |

* + - 1. UserSalary.cs
         1. The UserSalary.cs Model class should be written as shown below

The UserSalary.cs File

|  |
| --- |
| namespace DotnetAPI  {  public partial class UserSalary  {  public int UserId { get; set; }  public decimal Salary { get; set; }  public decimal AvgSalary { get; set; }  }  } |

* + 1. UserJobInfo.cs
       1. The UserJobInfo.cs Model class should be written as shown below

The UserJobInfo.cs File

|  |
| --- |
| namespace DotnetAPI  {  public partial class UserJobInfo  {  public int UserId { get; set; }  public string Department { get; set; }  public string JobTitle { get; set; }  public UserJobInfo()  {  if (Department == null) { Department = ""; }  if (JobTitle == null) { JobTitle = ""; }  }  }  } |

* 1. Now that we have our Users Model classes developed, we can start creating our User Controllers.

1. [User Controller](#TS10)

* 1. [The QS Class](#TS10).
     1. Before we begin, I wanted to show the QS (Query String) class I created so I don’t have to enter the SQL strings directly in the UserController.cs class
     2. QS provides static method to return these long strings so the UserController class isn’t so convoluted with messy SQL code.
     3. Below is the context of the QS class.

Contents of the QS Class

|  |
| --- |
| namespace DotnetAPI  {  public class QS  {  private static string CountRecordsString = "select \* from TutorialAppSchema.Users where UserId = ";  private static string DeleteUserString = "delete from TutorialAppSchema.Users where UserId = ";  private static string QueryUserString = "select UserId,FirstName,LastName,Email,Gender,Active from TutorialAppSchema.Users";  private static string QuerySingleUserString = "select UserId,FirstName,LastName,Email,Gender,Active from TutorialAppSchema.Users where UserId = ";  private static string QueryAllActiveRecordsString = "select u.UserId,u.FirstName,u.LastName,u.Email,u.Active,j.Department,  j.JobTitle,s.Salary,s.AvgSalary from TutorialAppSchema.Users u  left join TutorialAppSchema.UserJobInfo j on u.UserId = j.UserId  left join TutorialAppSchema.UserSalary s on u.UserId = s.UserId where active = 1";  private static string QueryAllInActiveRecordsString = "select u.UserId,u.FirstName,u.LastName,u.Email,u.Active,j.Department,  j.JobTitle,s.Salary,s.AvgSalary from TutorialAppSchema.Users u  left join TutorialAppSchema.UserJobInfo j on u.UserId = j.UserId  left join TutorialAppSchema.UserSalary s on u.UserId = s.UserId where active = 0";  public static string CountRecords()  {  return CountRecordsString;  }  public static string DeleteUser()  {  return DeleteUserString;  }  public static string QueryUser()  {  return QueryUserString;  }  public static string QuerySingleUser()  {  return QuerySingleUserString;  }  public static string QueryAllActiveRecords()  {  return QueryAllActiveRecordsString;  }  public static string QueryAllInActiveRecords()  {  return QueryAllInActiveRecordsString;  }  public static string UpdateUserSQL(int userId, string firstName, string lastName, string email, string gender, bool active)  {  return "update TutorialAppSchema.Users set [FirstName] = '" + firstName + "',[LastName] = '" + lastName + "',[Email] = '" + email + "',[Gender] = '" + gender + "',[Active] = '" + active + "' where [UserId] = '" + userId + "'";  }  public static string InsertUserSQL(string firstName, string lastName, string email, string gender, bool active)  {  return "insert into TutorialAppSchema.Users(FirstName,LastName,Email,Gender,Active) values ('" + firstName + "','" + lastName + "','" + email + "','" + gender + "','" + active + "')";  }  }  } |

* 1. [HttpGet](#TS10)
     1. Here we will be using the User class to retrieve (GET) data.
     2. We will be using two controllers to achieve there retrieval events.
     3. These controllers are where the requests come in directly from the browser.
     4. In our script below, we have two controllers. They are:
        1. GetUsers
           1. This returns all users in the Users table.
        2. GetSingleUser
           1. Returns one user based on the userId the browser submits.

UserController.cs File

|  |
| --- |
| using Microsoft.AspNetCore.Mvc;  namespace DotnetAPI.Controllers;  [ApiController]  [Route("[controller]")]  public class UserController : ControllerBase  {  DataContextDapper \_dapper;  // Without the builder statement in Program.cs file, we would need to ussue this command to get the config.  // IConfiguration config = new ConfigurationBuilder().AddJsonFile("appSettings.json").Build();  // It is Program.cs and more specifically the var builder that does this automatically for us now.  public UserController(IConfiguration config)  {  \_dapper = new DataContextDapper(config);  }  [HttpGet("GetUsers")]  public IEnumerable<User> GetUsers()  {  IEnumerable<User> users = \_dapper.LoadData<User>(QS.QueryUser());  return users;  }  [HttpGet("GetSingleUser/{userId}")]  public User GetSingleUser(int userId)  {  User user = \_dapper.LoadDataSingle<User>(QS.QuerySingleUser() + userId.ToString());  return user;  }  } |

* 1. [HttpPut and HttpPost](#TS10)
     1. Put is used for updating a user.
     2. Post will be used for adding a user.
     3. Below is the updated UserController.cs file with the put and post methods added.

UserController.cs File with all Action Methods Added

|  |
| --- |
| using Microsoft.AspNetCore.Mvc;  namespace DotnetAPI.Controllers;  [ApiController]  [Route("[controller]")]  public class UserController : ControllerBase  {  DataContextDapper \_dapper;  // Without the builder statement in Program.cs file, we would need to ussue this command to get the config.  // IConfiguration config = new ConfigurationBuilder().AddJsonFile("appSettings.json").Build();  // It is Program.cs and more specifically the var builder that does this automatically for us now.  public UserController(IConfiguration config)  {  \_dapper = new DataContextDapper(config);  }  [HttpGet("GetUsers")]  public IEnumerable<User> GetUsers()  {  IEnumerable<User> users = \_dapper.LoadData<User>(QS.QueryUser());  return users;  }  [HttpGet("GetSingleUser/{userId}")]  public User GetSingleUser(int userId)  {  User user = \_dapper.LoadDataSingle<User>(QS.QuerySingleUser() + userId.ToString());  return user;  }  [HttpPut]  public IActionResult EditUser(User user)  {  string sql = QS.UpdateUserSQL(user.UserId, user.FirstName, user.LastName, user.EMail, user.Gender, user.Active);  if (\_dapper.Execute(sql))  return Ok();  throw new Exception("Failed to update query.");  }  [HttpPost]  public IActionResult AddUser(User user)  {  string sql = QS.InsertUserSQL(user.FirstName, user.LastName, user.EMail, user.Gender, user.Active);  if (\_dapper.Execute(sql))  return Ok();  throw new Exception("Failed to insert new row.");  }  } |

* 1. [HttpDelete](#TS10)
     1. The following module describes how you can delete a user

Delete User Method in the UserController.cs file

|  |
| --- |
| [HttpDelete("DeleteUser/{userId}")]  public IActionResult DeleteUser(int userId)  {  string sql = QS.CountRecords() + userId.ToString();  Console.WriteLine("sql = [" + sql + "]");  if (\_dapper.Execute(sql))  {  sql = QS.DeleteUser() + userId.ToString();  if (\_dapper.Execute(sql))  return Ok();  throw new Exception("Failed to delete user.");  }  else  {  return NotFound();  }  } |

1. [DTO](#TS11)
   1. Data Transfer Objects provide the controllers objects the data feed from the database.
   2. Create a new folder called DTO.
   3. Copy the Users.cs file into the DTO file.
   4. Rename that newly copied file from Users.cs to UsersToAddDto.cs
   5. In the UsersToAddDto.cs file, perform these three steps:
      1. Rename the class to UsersToAddDto
      2. Rename the constructor to UsersToAddDto.
      3. Remove the public int UserId { get; set; } line.
   6. Finally, in our UserController.cs file where our Add user method exists, make this change to the Add method line:
      1. From: public IActionResult AddUser(User user)
      2. To: public IActionResult AddUser(UserToAddDto user)
   7. Now when the UserController executes in AddUser method, it will use the UsersToAddDto class as a template for taking in data.
      1. Since we don’t need the UserId as part of the input specifications from the user, UsersToAddDto will ensure it doesn’t expect the UserId first as input from the browser window.
   8. The takeaway here is DTO objects act as a filtering template for reading to and writing from the database.

1. [Namespaces](#TS12)
   1. In our application thus far, we are using four folders under the DotNetAPI directory:
      1. Controllers
      2. Data
      3. DTOs
      4. Model
   2. So far, the only folder we are actually assigning a sub namespace to is the Controllers items.
      1. For example, the namespace line in the UserController.cs file is namespace DotNetAPI.Controllers
   3. What we really want is the Data, DTOs and Model classes to follow suit and have their namespace set up the same way.
      1. For all the class files in the Data folder, we want to change the namespace line to read: namespace DotNetAPI.Data
      2. We will do the same for the Dtos and Models files.
   4. This will wreak havoc in the UserController.cs file because it will no longer be able to see the Data, DTO and Model classes.
   5. For this reason, we will add these three lines at the top of the UserController.cs file:

|  |
| --- |
| using DotnetAPI.Data;  using DotnetAPI.Dtos;  using DotnetAPI.Model; |

* 1. Now the UserController.cs file is happy.
  2. The takeaway here is we do not necessarily want to load everything into memory when the program initially loads.
     1. This way, each program will only load when the Controller file actually calls for it.

1. [Entity Framework](#TS13) [Setup](#TS13)
   1. Introduction
      1. This subject was covered in section C4 and C5 of this document if you require further information.
      2. Entity Framework is a method of writing to and retrieval of data from a database without having to issue any SQL commands.
      3. It is rigidly formatted with templates to ensure standardization of data.
   2. Requirements
      1. The Microsoft Entity Framework package need to be installed before beginning to code.
      2. At the PowerShell console, enter these two commands:
         1. dotnet add package Microsoft.EntityFrameworkCore
         2. dotnet add package Microsoft.EntityFrameworkCore.Relational
         3. dotnet add package Microsoft.EntityFrameworkCore.SqlServer
      3. Start the application back up with: dotnet watch run –launch-profile http
   3. Creating the DataContextEF.cs file
      1. In our data folder, create a file named DataContextEF.cs
      2. Below is the entire code for the file, which afterwards we will discuss each section in detail:

Code For the DataContextEF.cs File

|  |
| --- |
| using DotnetAPI.Model;  using Microsoft.EntityFrameworkCore;  namespace DotNetAPI.Data  {  public class DataContextEF : DbContext  {  private IConfiguration \_config;  public DataContextEF(IConfiguration config)  {  \_config = config;  }  protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)  {  if (!optionsBuilder.IsConfigured)  {  optionsBuilder  .UseSqlServer(\_config.GetConnectionString("DefaultConnectionString"),  optionsBuilder => optionsBuilder.EnableRetryOnFailure());  }  }  public virtual DbSet<User> Users { get; set; }  public virtual DbSet<User> UserSalary { get; set; }  public virtual DbSet<User> UserJobInfo { get; set; }  protected override void OnModelCreating(ModelBuilder modelBuilder)  {  modelBuilder.HasDefaultSchema("TutorialAppSchema");  modelBuilder.Entity<User>()  .ToTable("Users", "TutorialAppSchema")  .HasKey(u => u.UserId);  modelBuilder.Entity<UserSalary>()  .HasKey(u => u.UserId);  modelBuilder.Entity<UserJobInfo>()  .HasKey(u => u.UserId);  }  }  } |

* + 1. Now let’s talk about the individual sections of this file
       1. The class and constructor area:

|  |
| --- |
| public class DataContextEF : DbContext  {  private IConfiguration \_config;  public DataContextEF(IConfiguration config)  {  \_config = config;  } |

* + - * 1. Our class DataContextEF is inheriting all the methods from the DbContext class.
        2. IConfiguration \_config brings in the information from the appsettings.json file which is our database connection string.
        3. \_config will be our class instance variable that we use throughout the script.
      1. The OnConfiguring method:

|  |
| --- |
| protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)  {  if (!optionsBuilder.IsConfigured)  {  optionsBuilder  .UseSqlServer(\_config.GetConnectionString("DefaultConnectionString"),  optionsBuilder => optionsBuilder.EnableRetryOnFailure());  }  } |

* + - 1. OnConfiguring is a basic out-of-the-box method with a general template.
         1. We need to override certain aspects of this method to tailor it to our needs.
         2. What we are doing here is passing the DbContextOptionsBuilder object optionsBuilder to it.
         3. If for some reason it is not configured properly, we do so by telling it:

We are using SQLServer as our backend database service,

If we happen to have a SQL failure call at some point, retry the connection call.

* + - 1. Setting up our virtual references between the Class and their associated databases:

|  |
| --- |
| public virtual DbSet<User> Users { get; set; }  public virtual DbSet<User> UserSalary { get; set; }  public virtual DbSet<User> UserJobInfo { get; set; } |

* + - * 1. This is basically a cross reference between the Classes and their associated databases.
      1. Overriding OnModelCreating:

|  |
| --- |
| protected override void OnModelCreating(ModelBuilder modelBuilder)  {  modelBuilder.HasDefaultSchema("TutorialAppSchema");  modelBuilder.Entity<User>()  .ToTable("Users", "TutorialAppSchema")  .HasKey(u => u.UserId);  modelBuilder.Entity<UserSalary>()  .HasKey(u => u.UserId);  modelBuilder.Entity<UserJobInfo>()  .HasKey(u => u.UserId);  } |

* + - 1. So here is what is happening with the override.
         1. Entity Framework is a rigid, standardized framework that lays out default standards.

There standards include:

The Database table name ad its virtual class name relationship should be the same name.

All Primary key fields should be called Id.

When our environment does not exactly match these standards, we need to override the settings.

Above we are making four overrides.

Schema Name change

EF expects the schema name to be dbo.

We are telling EF the correct schema name is TutorialAppSchema.

Updates for the User DbSet.

The DbSet name is User. EF expects the SQL table to be the same.

.ToTable("Users", "TutorialAppSchema") is notifying EF the table name for User is actual Users

Since EF expects every key field to be Id, we notify EF that the key field is actually UserId

Updates for UserSalary.

Notify EF the key field name id UserId.

Updates for UserJobInfo

Notify EF the key field name id UserId.

* + 1. So that is the basic breakdown of our DataContextEF.cs file.
       1. We are simply tailoring it to our Needs.

1. [Entity Framework Controller](#TS14)
   1. In the Controllers folder, make a copy of the UserController.cs file and rename it UserEFController.cs
   2. In this file, also rename the class and constructor UserEFController.
   3. Below is a copy of the UserEFController.cs. Afterwards we will discuss it in detail:

The UserEFController.cs File

|  |
| --- |
| using AutoMapper;  using DotnetAPI.Data;  using DotnetAPI.Dtos;  using DotnetAPI.Model;  using Microsoft.AspNetCore.Mvc;  namespace DotnetAPI.Controllers;  [ApiController]  [Route("[controller]")]  public class UserEFController : ControllerBase  {  DataContextEF \_entityFramework;  IMapper \_mapper;  public UserEFController(IConfiguration config)  {  \_entityFramework = new DataContextEF(config);  \_mapper = new Mapper(new MapperConfiguration(cfg => {  cfg.CreateMap<UserToAddDto, User>();  }));  }  [HttpGet("GetUsers")]  // public IEnumerable<User> GetUsers()  public IEnumerable<User> GetUsers()  {  IEnumerable<User> users = \_entityFramework.Users.ToList<User>();  return users;  }  [HttpGet("GetSingleUser/{userId}")]  // public IEnumerable<User> GetUsers()  public User GetSingleUser(int userId)  {  User? user = \_entityFramework.Users.Where(u => u.UserId == userId).FirstOrDefault<User>();  if (user != null)  return user;  throw new Exception("User does not exist in database");  }    [HttpPut("EditUser")]  public IActionResult EditUser(User user)  {  User? userDb = \_entityFramework.Users.Where(u => u.UserId == user.UserId).FirstOrDefault<User>();    if (userDb != null)  {  userDb.Active = user.Active;  userDb.FirstName = user.FirstName;  userDb.LastName = user.LastName;  userDb.EMail = user.EMail;  userDb.Gender = user.Gender;  if (\_entityFramework.SaveChanges() > 0)  return Ok();  throw new Exception("Failed to Update User");  }  throw new Exception("User does not exist in database");  }  [HttpPost("AddUser")]  public IActionResult AddUser(UserToAddDto user)  {  User userDb = \_mapper.Map<User>(user);    \_entityFramework.Add(userDb);  if (\_entityFramework.SaveChanges() > 0)  return Ok();  throw new Exception("Failed to Add User");  }  [HttpDelete("DeleteUser/{userId}")]  public IActionResult DeleteUser(int userId)  {  User? userDb = \_entityFramework.Users.Where(u => u.UserId == userId).FirstOrDefault<User>();  if (userDb != null)  {  \_entityFramework.Users.Remove(userDb);  if (\_entityFramework.SaveChanges() > 0)  return Ok();  throw new Exception("Failed to Delete User");  }  throw new Exception("User does not exist in database");  }  } |

* 1. This script is broken down into seven sections:
     1. Definition as a controller

|  |
| --- |
| [ApiController]  [Route("[controller]")] |

* + 1. The class line and constructor area

|  |
| --- |
| public class UserEFController : ControllerBase  {  DataContextEF \_entityFramework;  IMapper \_mapper;  public UserEFController(IConfiguration config)  {  \_entityFramework = new DataContextEF(config);  \_mapper = new Mapper(new MapperConfiguration(cfg => {  cfg.CreateMap<UserToAddDto, User>();  }));  } |

* + - 1. ControllerBase extends its method to the class
      2. Two class definitions are made:
         1. DataContextEF \_entityFramework

\_entityFramework will be instantiated by the DataContextEF class and fed the database connection string through the appsettings.json file.

* + - * 1. IMapper \_mapper;

\_mapper will be instantiated via Mapper and a mapping configuration created which maps the UserToAddDto class properties to those in the User class.

* + 1. The GetUsers Method

|  |
| --- |
| [HttpGet("GetUsers")]  public IEnumerable<User> GetUsers()  {  IEnumerable<User> users = \_entityFramework.Users.ToList<User>();  return users;  } |

* + - 1. Provides an entry point in the URL line to retrieve a listing of all the users.
      2. Let’s analyze the \_entityFramework.Users.ToList<User>(); line
         1. Entity Framework is telling the Users SQL table to pull each rows of the table and instantiate it into an object of the User class.
         2. IEnumerable, being a methods which holds like objects, acts like a locomotive, carrying as many flatbed cars as there are records.

Each flatbed car will contains an instantiated object of the User class with its own properties for each rows of SQL data.

* + - * 1. Finally, we return the users object … ***but to where?***
        2. The answer is to the browser where the user requested the lookup to begin with.

Remember, the browser is an application just like Word or Excel, and it knows how to parse IEnumerable objects into displayable units on the screen.

* + 1. The GetSingleUser Method

|  |
| --- |
| [HttpGet(“GetSingleUser/{userId}”)]  // public IEnumerable<User> GetUsers()  public User GetSingleUser(int userId)  {  User? user = \_entityFramework.Users.Where(u => u.UserId == userId).FirstOrDefault<User>();  if (user != null)  return user;  throw new Exception(“User does not exist in database”);  } |

* + - 1. Here we are retrieving a single user.
      2. The user will supply the userId on the browser page (Either in a text box or dropdown selector).
      3. Let’s analyze this line: User? user = \_entityFramework.Users.Where(u => u.UserId == userId).FirstOrDefault<User>();
         1. First off, we are telling C# that user may be null by use of the ? after the User class.
         2. \_entityFramework.Users

Entity Framework is pulling down all the rows of the Users table.

* + - * 1. Where(u => u.UserId == userId).FirstOrDefault<User>();

Where is ingesting the lambda statement u => u.UserId == userId.

The u => u.UserId == userId is a concise way to define an anonymous function.

It takes an input **u** (representing each element in the Users collection) and evaluates whether the UserId property of u matches the value of userId.

In other words, there will be as many **u** elements as there are records, each one saying either *Yep, found a match* or *no match found*.

In this context, u => u.UserId == userId is almost acting like a *ForEach* loop, analyzing each row for a match.

If a **u** element finds a match, Entity Framework will invoke the FirstorDefault method to pull that record if one is found.

* + - * 1. FirstOrDefault<User>();

The first matching record that is found will take the column data from that record and load it into an instance of the User class.

* + - * 1. If user is not null (A record match was successful), the user object will be returned.
      1. When this statement is done running, either the user object will be populated with one record of data, or it will be null because no matches were found.
         1. A successful record search will return a code 200, which an unsuccessful search will return a code 404.
      2. Finally the user object is returned to the browser.
    1. Edit a User

|  |
| --- |
| [HttpPut("EditUser")]  public IActionResult EditUser(User user)  {  User? userDb = \_entityFramework.Users.Where(u => u.UserId == user.UserId).FirstOrDefault<User>();    if (userDb != null)  {  userDb.Active = user.Active;  userDb.FirstName = user.FirstName;  userDb.LastName = user.LastName;  userDb.EMail = user.EMail;  userDb.Gender = user.Gender;  if (\_entityFramework.SaveChanges() > 0)  return Ok();  throw new Exception("Failed to Update User");  }  throw new Exception("User does not exist in database");  } |

* + - 1. I am not going to cover the User userDb line as we went over this extensively in the previous section.
      2. If the record is not null, the browser will allow the user to enter new values in each field.
      3. Once the user clicks on the Execute button, this entire method fired off.
      4. Notice in our if statement where we load the userDB objects with the fields provided by the user.
      5. Entity Framework then executes the SaveChanges method and there new values replace those existing in that particular row.
    1. Add a User

|  |
| --- |
| [HttpPost("AddUser")]  public IActionResult AddUser(UserToAddDto user)  {  User userDb = \_mapper.Map<User>(user);    \_entityFramework.Add(userDb);  if (\_entityFramework.SaveChanges() > 0)  return Ok();  throw new Exception("Failed to Add User");  } |

* + - 1. Since I have covered every aspect of these methods so far, the only thing we will talk about here is the using of AutoMapper.
      2. Do you remember this syntax is the second part of our discussion:

|  |
| --- |
| \_mapper = new Mapper(new MapperConfiguration(cfg => {  cfg.CreateMap<UserToAddDto, User>(); |

* + - 1. In this AddUser method we have the line:
         1. User userDb = \_mapper.Map<User>(user);
      2. Basically what happens here is the data in the User fields where the user entered data is transmitted automatically to the fields in the UserToAddDto class.
      3. Then those fields are saved to disk, thus adding a new record.
    1. Delete a Record

|  |
| --- |
| [HttpDelete("DeleteUser/{userId}")]  public IActionResult DeleteUser(int userId)  {  User? userDb = \_entityFramework.Users.Where(u => u.UserId == userId).FirstOrDefault<User>();  if (userDb != null)  {  \_entityFramework.Users.Remove(userDb);  if (\_entityFramework.SaveChanges() > 0)  return Ok();  throw new Exception("Failed to Delete User");  }  throw new Exception("User does not exist in database");  } |

* + - 1. Since we have covered so many topics in the previous methods, I am hoping you can see exactly how this one works.
      2. Basically a User ID is passed to Entity Frameworks. EF checks to see if it exists.
      3. If it does, it deletes the record.

1. [The User Repository and User Repository Interface](#TS15)
   1. The User Repository
      1. A User Repository is a specific implementation of the Repository Design Pattern that focuses on managing and abstracting data access for user-related entities in an application.
      2. It acts as a mediator between the data access layer (e.g., database) and the business logic layer, ensuring clean separation of concerns.
      3. Key Features of a User Repository:
         1. Encapsulation of Data Access Logic:
            1. It centralizes all user-related database operations, such as retrieving, adding, updating, or deleting user records.
         2. Abstraction:
            1. Provides an interface to interact with user data without exposing the underlying database or ORM (e.g., Entity Framework).
         3. Reusability:
            1. Promotes code reuse by offering a consistent way to handle user data across the application.
         4. Testability:
            1. Makes unit testing easier by allowing the use of mock repositories.
      4. How does the User Repository interact with the way things are working now?
         1. In our current setup, we have the Controllers talking directly to the EF Data Context.
            1. In other words, when a request comes in from the browser to the controller, that controller interacts directly with the database objects.
            2. If a request came in from the controller to get a listing of all users via the GetUsers method, it would also have access to the AddUser, EditUser and DeleteUser methods.
            3. What we want is for this request to only have access to the GetUsers method.
         2. How we can achieve this is creating a repository that contains all the methods and exposing the controller to that User Repository.
            1. However, if this User Repository also has access to all the methods, what’s the point? We’re right back to where we started.
            2. If you remember, an interface is a file which contains abstract methods.

Its purpose is to provide governance to the concrete class as to which methods will be activated upon instantiation.

* + - * 1. If we had a way to only have the GetUsers method activated when a GetUsers call was made from the controller, and deactivate the other methods, that would be wonderful!
        2. That is where the IUserInterface comes in. We will talk about that next.
  1. The User Repository Interface
     1. Ok, why have a User Repository Interface in the first place? Isn’t the User Interface along just fine.
     2. Let me give you an analogy.
        1. In any given month you pay 4 bills, those being:
           1. Rent
           2. Electric
           3. Cable
           4. Cell Phone
        2. At your bank, you have set up for savings accounts for paying bills:
           1. Rent Savings account
           2. Electric Savings account
           3. Cable Savings account
           4. Cell Phone Savings account.
        3. You prefer paying your bills with cash, so each time you pay a bill, you go to the ATM machine to withdraw month from the particular savings account associated to the bill you are paying.
           1. This ATM machine has access to all 4 of your savings accounts.
           2. When you go to withdraw money, all 4 of the savings accounts show up on the screen.
           3. That means you could accidently withdraw money from the Rent Savings account when you were getting cash to pay the electric bill .
           4. But this ATM machine is a special one. You can enter the account number of the bill and our ATM machine will know exactly what savings account needs to be withdrawn from.
           5. At the ATM machine, you enter your electric bill account number.
           6. The ATM machine recognizes this as your electric bill account.
           7. When the selection screen comes up to withdraw cash from a particular account, *only the Electric Savings account option comes up as an account to withdraw from*.
           8. The other savings account withdraw buttons are not displayed.
           9. The same thing will happen when paying the other 4 bills. Only their associated savings accounts will appear on the withdrawal screen
           10. Now you will never accidently withdraw from the wrong account!
     3. So how do we apply this concept to this project? Here are the four steps to achieve this process:
        1. First we need to add a Scope of the IUserRepository and UserRepository classes into the Builder Services in the Program.cs file.
        2. Next, we move all the current methods of accessing the Entity Framework DataContextEF.cs file out of the UserEFController.cd file and put them into the UserRepository.cs file.
           1. We do this because it will be the UserRepository.cs file that directly talks to the database now.
           2. We also need to update the UserRepository.cs file to take in IUserRepository.cs as an interface file
        3. Thirdly, we create our IUserRepository.cs file which will only contains the abstract bodyless methods used in the UserRepository.cs file.
        4. Lastly, we update our UserEFController.cs file to call the IUserRepository.cs functions when a controller method is executed.
     4. Basically we are inserting the UserRepository between the Controller and the Entity Framework Data Context.
     5. The diagram below depicts how the User Repository acts a liaison between the Controller and data, thus preventing direct access of information to the browser.

Implementation of the User Repository

Data flow before implementing the User Repository Data flow after implementing the User Repository

|  |  |
| --- | --- |
| Controller EF Data Context | Controller IUserRepository UserRepository EF Data Context |

* 1. The repository flow:
     1. Abstracts the layer of data away from the user.
  2. Setting up the User Repository.
     1. To set up our repository for use, we will need to do three things:
        1. Update the Program.cs file to add a Scoped service to allow IUserRepository to work with UserRepository.
        2. Create the UserRepository.cs file
        3. Create the IUserRepository.cs file.
        4. Update our EF Controllers to communicate with the IUserRepository instead of directly with the Data Context.
     2. These four changes are described below:  
        1. Update the Program.cs file to add a Scoped service to allow IUserRepository to work with UserRepository.
           1. We need to tell our app model that the IUserRepository and UserRepository classes should be wrapped up into the application.
           2. Place this line just above var app = builder.Build() in the Program.cs file:

Builder.Services.AddScoped<IUserRepository,UserRepository>();

* + - 1. Create the UserRepository.cs file
         1. Inside the data folder, create a new file called UserRepository.cs
         2. All of the code that used to be in the UserEFController.cs file is now in this file.
         3. One thing of note about this code is in the AddEntity and RemoveEntity methods.

First, what is an entity?

It is a generic term that refers to adding or removing data from any one of our three tables.

Remember that we have three database tables:

Users

UserSalary

UserJobInfo

When we add a new user, not only will we add them to the Users table, but we also have to add their associated information to the other two tables as well.

The use of Generics with Entities

Consider these two methods in our UserRepository.cs file:

The AddEntity Method

|  |
| --- |
| public void AddEntity<T>(T entityToAdd)  {  if (entityToAdd != null)  {  \_entityFramework.Add(entityToAdd);  }  } |

The RemoveEntity Method

|  |
| --- |
| public void RemoveEntity<T>(T entityToDelete)  {  if (entityToAdd != null)  {  \_entityFramework.Remove(entityToDelete);  }  } |

They are method to add and remove records to/from any of our three tables.

Now, we could have made three methods for adding records to the Users, UserSalary and UserJobInfo tables.

That would have been a lot of unnecessary code.

Instead we use Generic methods. These methods are made generic by adding the <T> symbol after the method name.

In our Controller class, when we add a user to any of these three tables, we simply call the AddEntity method.

Within the UserEFController.cs file, we have three methods to both add and delete records to/from each of the three tables.

UserEFController Add and Remove Commands for Each of the Three Tables

|  |  |  |
| --- | --- | --- |
| Action | Table Name | Controller Command |
| Add | Users | \_userRepository.AddEntity<User>(userDb); |
| Add | UserSalary | \_userRepository.AddEntity<UserSalary>(userForInsert); |
| Add | UserJobInfo | \_userRepository.AddEntity<UserJobInfo>(userToDelete); |
| Delete | Users | \_userRepository.RemoveEntity<User>(userToDelete); |
| Delete | UserSalary | \_userRepository.RemoveEntity<UserSalary>(userForInsert); |
| Delete | UserJobInfo | \_userRepository.RemoveEntity<UserJobInfo>(userToDelete); |

When these commands they prospective method (Add or Delete), the data set in <> is passed to the AddEntity or RemoveEntity method.

In the AddEntity method, the syntax (T entityToAdd) uses the T to pick up the Class name (User, UserSalary, UserJobInfo).

So my using generics in the AddEntity and RemoveEntity classes, one method can address all the table adds. The same is true for the removes.

* + - * 1. Write the following code into the UserRepository.cs file.

UserRepository.cs File

|  |
| --- |
| using DotnetAPI.data;  using DotnetAPI.Model;  namespace DotnetAPI.Data  {  public class UserRepository : IUserRepository  {  DataContextEF \_entityFramework;  public UserRepository(IConfiguration config)  {  \_entityFramework = new DataContextEF(config);  }  public bool SaveChanges()  {  return \_entityFramework.SaveChanges() > 0;  }  // public bool AddEntity<T>(T entityToAdd)  public void AddEntity<T>(T entityToAdd)  {  if (entityToAdd != null)  {  \_entityFramework.Add(entityToAdd);  // return true;  }  // return false;  }  // public bool RemoveEntity<T>(T entityToRemove)  public void RemoveEntity<T>(T entityToRemove)  {  if (entityToAdd != null)  {  \_entityFramework.Remove(entityToRemove);  // return true;  }  // return false;  }  public IEnumerable<User> GetUsers()  {  IEnumerable<User> users = \_entityFramework.Users.ToList<User>();  return users;  }  public User GetSingleUser(int userId)  {  User? user = \_entityFramework.Users  .Where(u => u.UserId == userId)  .FirstOrDefault<User>();  if (user != null)  {  return user;  }    throw new Exception("Failed to Get User");  }  public UserSalary GetSingleUserSalary(int userId)  {  UserSalary? userSalary = \_entityFramework.UserSalary  .Where(u => u.UserId == userId)  .FirstOrDefault<UserSalary>();  if (userSalary != null)  {  return userSalary;  }    throw new Exception("Failed to Get User");  }  public UserJobInfo GetSingleUserJobInfo(int userId)  {  UserJobInfo? userJobInfo = \_entityFramework.UserJobInfo  .Where(u => u.UserId == userId)  .FirstOrDefault<UserJobInfo>();  if (userJobInfo != null)  {  return userJobInfo;  }    throw new Exception("Failed to Get User");  }  }  } |

* + - 1. Create the IUserRepository.cs file
         1. These interfaces work just like normal interfaces such as only giving the Hourly payroll process to the hourly methods in a Payroll class.
         2. So make a new file in the Data folder called IUserRepository.cs
         3. It should look like this

The IUserRepository.cs file

|  |
| --- |
| using DotnetAPI.Model;  namespace DotnetAPI.data  {  public interface IUserRepository  {  public bool SaveChanges();  public void AddEntity<T>(T entityToAdd);  public void RemoveEntity<T>(T entityToAdd);  public IEnumerable<User> GetUsers();  public User GetSingleUser(int userId);  public UserSalary GetSingleUserSalary(int userId);  public UserJobInfo GetSingleUserJobInfo(int userId);  }  } |

* + - 1. Update our UserEFController.cs file to allow it is use IUserRepository

The UserEFController.cs File

|  |
| --- |
| using AutoMapper;  using DotnetAPI.data;  using DotnetAPI.Dtos;  using DotnetAPI.Model;  using Microsoft.AspNetCore.Mvc;  namespace DotnetAPI.Controllers;  [ApiController]  [Route("[controller]")]  public class UserEFController : ControllerBase  {  IUserRepository \_userRepository;  IMapper \_mapper;  public UserEFController(IConfiguration config, IUserRepository userRepository)  {  \_userRepository = userRepository;  \_mapper = new Mapper(new MapperConfiguration(cfg =>{  cfg.CreateMap<UserToAddDto, User>();  cfg.CreateMap<UserSalary, UserSalary>().ReverseMap();  cfg.CreateMap<UserJobInfo, UserJobInfo>().ReverseMap();  }));  }  [HttpGet("GetUsers")]  // public IEnumerable<User> GetUsers()  public IEnumerable<User> GetUsers()  {  IEnumerable<User> users = \_userRepository.GetUsers();  return users;  }  [HttpGet("GetSingleUser/{userId}")]  // public IEnumerable<User> GetUsers()  public User GetSingleUser(int userId)  {  return \_userRepository.GetSingleUser(userId);  }    [HttpPut("EditUser")]  public IActionResult EditUser(User user)  {  User? userDb = \_userRepository.GetSingleUser(user.UserId);    if (userDb != null)  {  userDb.Active = user.Active;  userDb.FirstName = user.FirstName;  userDb.LastName = user.LastName;  userDb.EMail = user.EMail;  userDb.Gender = user.Gender;  if (\_userRepository.SaveChanges())  {  return Ok();  }  throw new Exception("Failed to Update User");  }    throw new Exception("Failed to Get User");  }  [HttpPost("AddUser")]  public IActionResult AddUser(UserToAddDto user)  {  User userDb = \_mapper.Map<User>(user);    \_userRepository.AddEntity<User>(userDb);  if (\_userRepository.SaveChanges())  {  return Ok();  }  throw new Exception("Failed to Add User");  }  [HttpDelete("DeleteUser/{userId}")]  public IActionResult DeleteUser(int userId)  {  User? userDb = \_userRepository.GetSingleUser(userId);    if (userDb != null)  {  \_userRepository.RemoveEntity<User>(userDb);  if (\_userRepository.SaveChanges())  {  return Ok();  }  throw new Exception("Failed to Delete User");  }    throw new Exception("Failed to Get User");  }  [HttpGet("UserSalary/{userId}")]  public UserSalary GetUserSalaryEF(int userId)  {  return \_userRepository.GetSingleUserSalary(userId);  }  [HttpPost("UserSalary")]  public IActionResult PostUserSalaryEf(UserSalary userForInsert)  {  \_userRepository.AddEntity<UserSalary>(userForInsert);  if (\_userRepository.SaveChanges())  {  return Ok();  }  throw new Exception("Adding UserSalary failed on save");  }  [HttpPut("UserSalary")]  public IActionResult PutUserSalaryEf(UserSalary userForUpdate)  {  UserSalary? userToUpdate = \_userRepository.GetSingleUserSalary(userForUpdate.UserId);  if (userToUpdate != null)  {  \_mapper.Map(userForUpdate, userToUpdate);  if (\_userRepository.SaveChanges())  {  return Ok();  }  throw new Exception("Updating UserSalary failed on save");  }  throw new Exception("Failed to find UserSalary to Update");  }  [HttpDelete("UserSalary/{userId}")]  public IActionResult DeleteUserSalaryEf(int userId)  {  UserSalary? userToDelete = \_userRepository.GetSingleUserSalary(userId);  if (userToDelete != null)  {  \_userRepository.RemoveEntity<UserSalary>(userToDelete);  if (\_userRepository.SaveChanges())  {  return Ok();  }  throw new Exception("Deleting UserSalary failed on save");  }  throw new Exception("Failed to find UserSalary to delete");  }  [HttpGet("UserJobInfo/{userId}")]  public UserJobInfo GetUserJobInfoEF(int userId)  {  return \_userRepository.GetSingleUserJobInfo(userId);  }  [HttpPost("UserJobInfo")]  public IActionResult PostUserJobInfoEf(UserJobInfo userForInsert)  {  \_userRepository.AddEntity<UserJobInfo>(userForInsert);  if (\_userRepository.SaveChanges())  {  return Ok();  }  throw new Exception("Adding UserJobInfo failed on save");  }  [HttpPut("UserJobInfo")]  public IActionResult PutUserJobInfoEf(UserJobInfo userForUpdate)  {  UserJobInfo? userToUpdate = \_userRepository.GetSingleUserJobInfo(userForUpdate.UserId);  if (userToUpdate != null)  {  \_mapper.Map(userForUpdate, userToUpdate);  if (\_userRepository.SaveChanges())  {  return Ok();  }  throw new Exception("Updating UserJobInfo failed on save");  }  throw new Exception("Failed to find UserJobInfo to Update");  }  [HttpDelete("UserJobInfo/{userId}")]  public IActionResult DeleteUserJobInfoEf(int userId)  {  UserJobInfo? userToDelete = \_userRepository.GetSingleUserJobInfo(userId);  if (userToDelete != null)  {  \_userRepository.RemoveEntity<UserJobInfo>(userToDelete);  if (\_userRepository.SaveChanges())  {  return Ok();  }  throw new Exception("Deleting UserJobInfo failed on save");  }  throw new Exception("Failed to find UserJobInfo to delete");  }  } |

* + 1. Now our IUserRepository has access to our UserRepository.
       1. To summarize, we add the Repositories as a means to only expose those methods needed for a particular action.

1. Authorization Table in SQL
   1. The purpose of this section is to has the user register their email and password. Then this information will be used to log in.
      1. We will be performing three steps in this section:
         1. Creating the Auth table which holds the PasswordHash and PasswordSalt values
         2. Updating the appsettings.json file to add the appsettings Password key.
         3. Adding three DTO Model classes for Registration, Login and Login confirmation.
   2. Here we will be building a password table in SQL that will intake a password hash and a password salt.
      1. We will be encoding our password within the SQL table.
   3. First lets create our Auth table in SQL
      1. create table TutorialAppSchema.Auth(Email nvarchar(50), PasswordHash varbinary(max), PasswordSalt varbinary(max))
   4. Next we need to update the appsettings.json file to add the auth settings. The additions will be in blue below:

The appsettings.json File

|  |
| --- |
| {  “ConnectionStrings”: {  “DefaultConnection”:”Server=DAVES\_PC;Database=DotNetCourseDatabase;TrustServerCertificate=true;Trusted\_Connection=true;”  // “DefaultConnection”:”Server=DAVES\_PC;Database=DotNetCourseDatabase;Trusted\_Connection=true;User ID=srv\_apps;Password=Frankenmuth.;”  },  “Logging”: {  “LogLevel”: {  “Default”: “Information”,  “Microsoft.AspNetCore”: “Warning”  }  },  “AllowedHosts”: “\*”,  “AppSettings”: {  “PasswordKey”: “knjjfjf-kfjfpojpogpg-kjdgh48750dngoidut0843-knfjsdhgf-98kjfisdjgs-skfsjgdjgsdgjk”  } |

* 1. Next, we will create three new DTOs models which are:
     1. Registration
        1. The user registers the first time to get registered.
     2. Login
        1. This is where the user issues their username and password.
     3. Login Confirmation
        1. This compares the users password against what is in the database
  2. The Registration DTO Model file
     1. Create a file named UserForRegistrationDto.cs and fill it with this text:

UserForRegistrationDto.cs File

|  |
| --- |
| namespace DotnetAPI.Dtos  {  partial class UserForRegistrationDto  {  string Email { get; set; }  string Password { get; set; }  string PasswordConfirm { get; set; }  public UserForRegistrationDto()  {  if (Email == null)  {  Email = “”;  }  if (Password == null)  {  Password = “”;  }  if (PasswordConfirm == null)  {  PasswordConfirm = “”;  }  }  }  } |

* 1. The Login DTO Model file
     1. Create a file named UserForLoginDto.cs and put this text in it.

UserForLoginDto.cs File

|  |
| --- |
| namespace DotnetAPI.Dtos  {  partial class UserForLoginDto  {  string Email { get; set; }  string Password { get; set; }  string PasswordConfirm { get; set; }  public UserForLoginDto()  {  if (Email == null)  {  Email = “”;  }  if (Password == null)  {  Password = “”;  }  }  }  } |

* + 1. The Login Confirmation DTO Model file
       1. Create a file named UserForLoginConfirmationDto.cs and put this text in it:

UserForLoginConfirmationDto.cs File

|  |
| --- |
| using Microsoft.AspNetCore.Identity;  namespace DotnetAPI.Dtos  {  partial class UserForLoginConfirmationDto  {  byte[] PasswordHash { get; set; }  byte[] PasswordSalt { get; set; }  public UserForLoginConfirmationDto()  {  {  if (PasswordHash == null)  {  PasswordHash = new byte[0];  }    if (PasswordSalt == null)  {  PasswordSalt = new byte[0];  }  }  }  }  } |

1. Setting up the Auth controller
   1. Here we will set up the Auth controller and a registration process so we can store their password.
   2. Create a new controller
      1. Name it AuthController.cs
      2. Here are the contents you need to type in:

AuthController.cs File

|  |
| --- |
| using System.Data;  using System.Security.Cryptography;  using System.Text;  using DotnetAPI.Data;  using DotnetAPI.Dtos;  using Microsoft.AspNetCore.Cryptography.KeyDerivation;  using Microsoft.AspNetCore.Mvc;  using Microsoft.Data.SqlClient;  namespace DotnetAPI.Controllers  {  public class AuthController : ControllerBase  {  // \_dapper becomes the object with all the built-in SQL connections and methods  private readonly DataContextDapper \_dapper;  // \_config beings in the appsettings.json data connection and password information  private readonly IConfiguration \_config;  public AuthController(IConfiguration config)  {  // Make both these variables instance variables so they are accessible throughout the entire program.  \_dapper = new DataContextDapper(config);  \_config = config;  }  [HttpPost("Register")]  public IActionResult Register(UserForRegistrationDto userForRegistration)  {  // Make sure the password and password confirmation entries match  if (userForRegistration.Password == userForRegistration.PasswordConfirm)  {  // Make sure the user isn’t already registered  string sqlCheckUserExists = "SELECT Email FROM TutorialAppSchema.Auth WHERE Email = '" +  userForRegistration.Email + "'";    // Count how many users in the Auth table match the registering user’s E-Mail address.  IEnumerable<string> existingUsers = \_dapper.LoadData<string>(sqlCheckUserExists);  if (existingUsers.Count() == 0)  {  // Fill at 16 byte (128 bit) bite array with random numbers for the salt randomizing  byte[] passwordSalt = new byte[128 / 8];  using (RandomNumberGenerator rng = RandomNumberGenerator.Create())  {  rng.GetNonZeroBytes(passwordSalt);  }  // Get the password which was scrambled and encrypted by the GetPasswordHash function  byte[] passwordHash = GetPasswordHash(userForRegistration.Password, passwordSalt);  // Insert the e-mail, hash and salt items in the Auth database.  string sqlAddAuth = @"  INSERT INTO TutorialAppSchema.Auth ([Email],  [PasswordHash],  [PasswordSalt]) VALUES ('" + userForRegistration.Email +  "', @PasswordHash, @PasswordSalt)";  List<SqlParameter> sqlParameters = new List<SqlParameter>();  SqlParameter passwordSaltParameter = new SqlParameter("@PasswordSalt", SqlDbType.VarBinary);  passwordSaltParameter.Value = passwordSalt;  SqlParameter passwordHashParameter = new SqlParameter("@PasswordHash", SqlDbType.VarBinary);  passwordHashParameter.Value = passwordHash;  sqlParameters.Add(passwordSaltParameter);  sqlParameters.Add(passwordHashParameter);  if (\_dapper.ExecuteSqlWithParameters(sqlAddAuth, sqlParameters))  {  string sqlAddUser = @"  insert into TutorialAppSchema.Users(  [FirstName],  [LastName],  [Email],  [Gender],  [Active]  ) values (" +  "'" + userForRegistration.FirstName +  "', '" + userForRegistration.LastName +  "', '" + userForRegistration.Email +  "', '" + userForRegistration.Gender +  "', 1)";  if (\_dapper.ExecuteSql(sqlAddUser))  {  return Ok();  }  throw new Exception("Failed to user");  }  throw new Exception("Failed to register user.");  }  throw new Exception("User with this email already exists!");  }  throw new Exception("Passwords do not match!");  }  [HttpPost("Login")]  public IActionResult Login(UserForLoginDto userForLogin)  {  string sqlForHashAndSalt = @"SELECT  [PasswordHash],  [PasswordSalt] FROM TutorialAppSchema.Auth WHERE Email = '" +  userForLogin.Email + "'";  UserForLoginConfirmationDto userForConfirmation = \_dapper  .LoadDataSingle<UserForLoginConfirmationDto>(sqlForHashAndSalt);  byte[] passwordHash = GetPasswordHash(userForLogin.Password, userForConfirmation.PasswordSalt);  // if (passwordHash == userForConfirmation.PasswordHash) // Won't work  for (int index = 0; index < passwordHash.Length; index++)  {  if (passwordHash[index] != userForConfirmation.PasswordHash[index]){  return StatusCode(401, "Incorrect password!");  }  }  return Ok();  }  private byte[] GetPasswordHash(string password, byte[] passwordSalt)  {  string passwordSaltPlusString = \_config.GetSection("AppSettings:PasswordKey").Value +  Convert.ToBase64String(passwordSalt);  return KeyDerivation.Pbkdf2(  password: password,  salt: Encoding.ASCII.GetBytes(passwordSaltPlusString),  prf: KeyDerivationPrf.HMACSHA256,  iterationCount: 1000000,  numBytesRequested: 256 / 8  );  }  }  } |

* + 1. Notice I put ExecuteSqlWithParameters in blue above?
       1. That is because we had to add a new method to our DataContextDapper.cs file
       2. I want to show a comparison between this new method and a similar method in PowerShell that also writes to the same database.
          1. I have color coded the similarities in the methods across both C# and PowerShell:

New Method in the DataContextDapper.cs File

|  |
| --- |
| public bool ExecuteSqlWithParameters(string sql, List<SqlParameter> parameters)  {  SqlCommand commandWithParams = new SqlCommand(sql);  foreach (SqlParameter parameter in parameters)  {  commandWithParams.Parameters.Add(parameter);  }  SqlConnection dbConnection = new SqlConnection(\_config.GetConnectionString("DefaultConnection"));  dbConnection.Open();  commandWithParams.Connection = dbConnection;  int rowsAffected = commandWithParams.ExecuteNonQuery();  dbConnection.Close();  return rowsAffected > 0;  } |

* + - 1. Now here is a similar method in PowerShell.
         1. The takeaway here is how similar these two programming languages are in syntax when it comes to writing to SQL.
         2. This is because they are basically doing the same thing:

PowerShell function that writes to the same SQL table

|  |
| --- |
| function ExecuteSqlWithParameters {  Param (  [string]$sql  )  $connectionString="DSN=DotNetCourseDatabase;"  $dbConnection = New-Object System.Data.Odbc.OdbcConnection  $dbConnection.ConnectionString = $connectionString  $dbConnection.open()  $commandWithParams = new-object System.Data.Odbc.OdbcCommand($sql,$dbConnection)  $rowsAffected = $commandWithParams.ExecuteNonQuery()  $dbConnection.close()  return $rowsAffected -gt 0  } |

* + - 1. They are almost identical! That is because under the covers they are doing the exact same thing.

1. JWT Token Validation.
   1. Install this package: dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
   2. In the Program.cs file, add these lines just above the var app = builder.Build(); line:

|  |
| --- |
| string? tokenKeyString = builder.Configuration.GetSection("AppSettings:TokenKey").Value;  SymmetricSecurityKey tokenKey = new SymmetricSecurityKey(  Encoding.UTF8.GetBytes(  tokenKeyString != null ? tokenKeyString : ""));  TokenValidationParameters tokenValidationParameters = new TokenValidationParameters()  {  IssuerSigningKey = tokenKey,  ValidateIssuer = false,  ValidateIssuerSigningKey = false,  ValidateAudience = false  };  builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)  .AddJwtBearer(options =>  {  options.TokenValidationParameters = tokenValidationParameters;  }); |