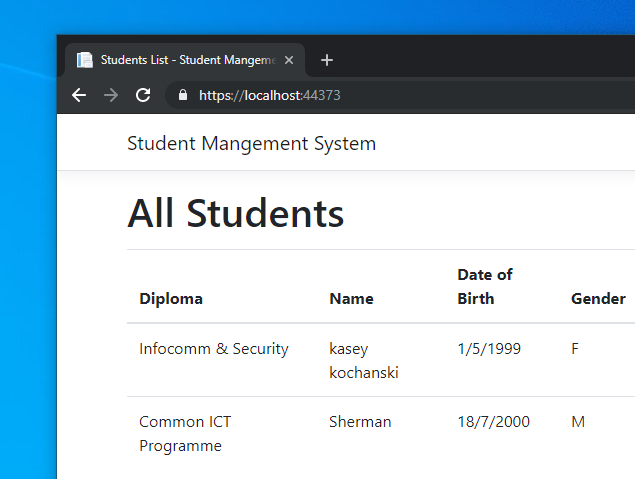
# About this practical

Throughout this practical, we will be building a ASP.NET application that utilises Entity Framework Core for database access.

This practical consist of 2 parts, lab 1 and lab 2. Lab 1 mainly focusing on initial project setup and data model building while lab 2 will focus on interacting with database using our data model.

The web application we are building is a simple student management system that manages student’s personal details, diploma info and course modules.

The source code for the practical demo can be found in Github: <https://github.com/c0j0s/StudentManagementSystem>

All steps can be view separately in corresponding branches, master branch is the complete version.

## Prerequisites:

Latest version of Visual Studio 2019 or NuGet client version 3.6.0.

SQL studio management 18.2

Contents

[About this practical 1](#_Toc16152483)

[Prerequisites: 1](#_Toc16152484)

[1. Setting up the Project 2](#_Toc16152485)

[2. Installing Entity Framework Core 3](#_Toc16152486)

[3. Code First Approach: 4](#_Toc16152487)

[3.1. Build Your Data Model 4](#_Toc16152488)

[3.2. Define Relationships between Entities 6](#_Toc16152489)

[3.3. Define Constrains Using Data Annotation 9](#_Toc16152490)

[3.4. Data Seeding 11](#_Toc16152491)

[3.5. Generate the Database 12](#_Toc16152492)

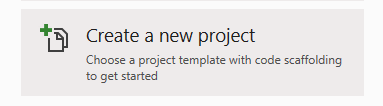
[4. Database First Approach: 14](#_Toc16152493)

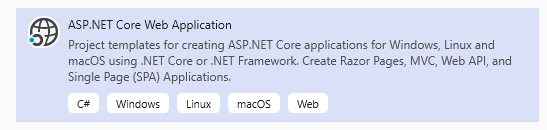
[4.1. Add a New Project in Solution 14](#_Toc16152494)

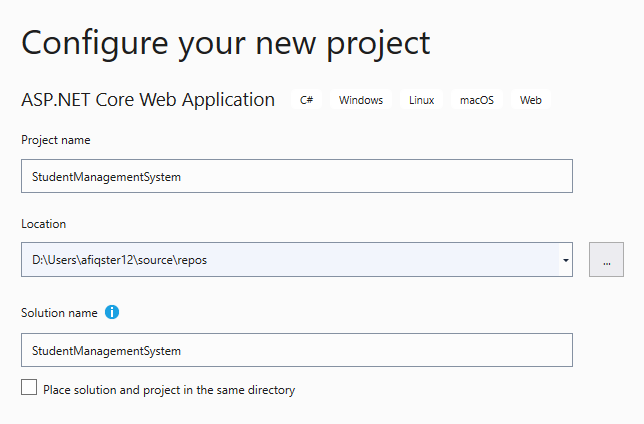
[4.2. Reverse Engineer/Scaffold Existing Database 16](#_Toc16152495)

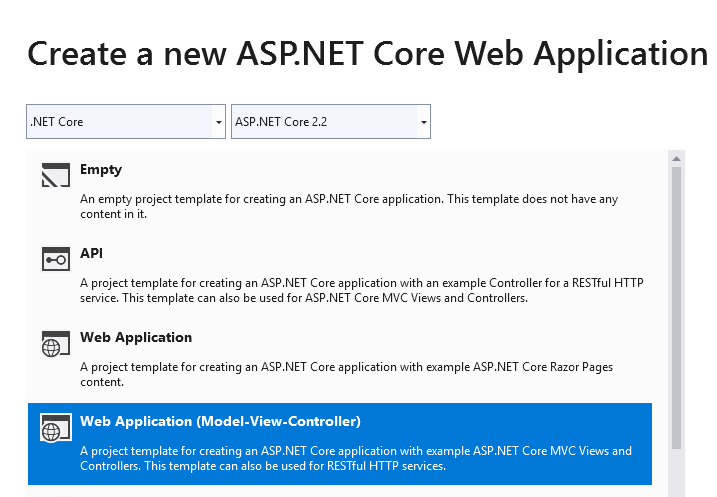
[4.3. Take a look at the Generated Files 17](#_Toc16152496)

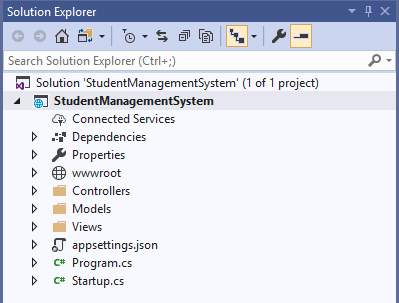
# Setting up the Project

* 1. When you launch Visual Studio 2019, you will be greeted with this screen, choose “**Create a new project**” to proceed.  
     
  2. Choose the follow application type, this will create the necessary project structure for our project.



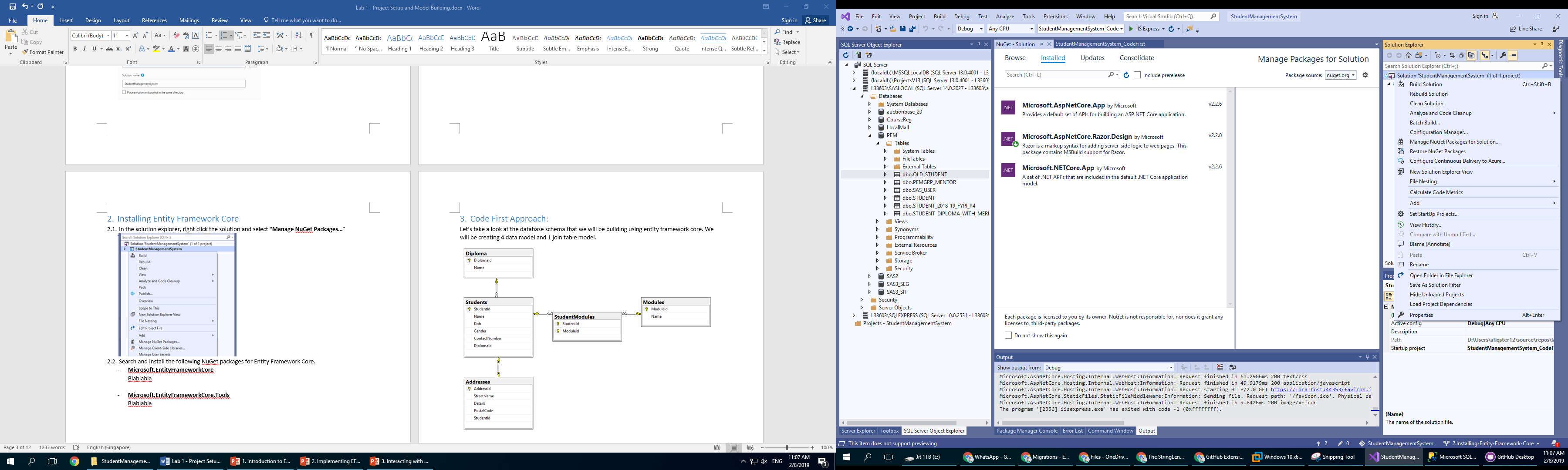
* 1. The next page will configure the project name and location of source codes. Here we enter our project name as “**StudentManagementSystem**” and click “Create” to proceed.
  2. The next page will allow us to choose what project template to begin with, we will be using “**Web Application (Model-View-Controller)**”. Click “Next” to proceed.



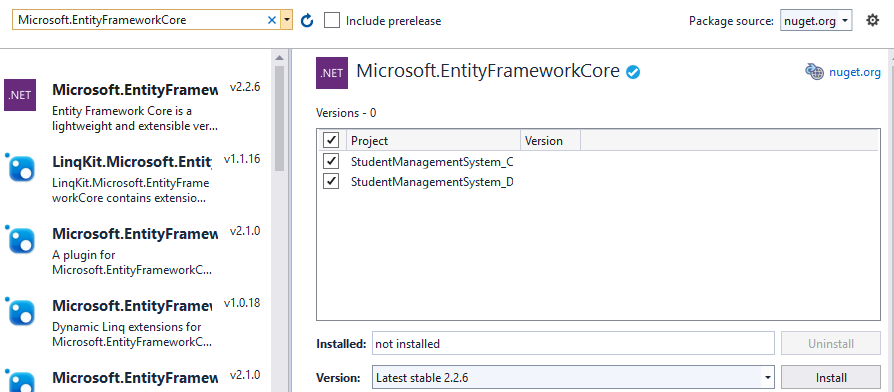
* 1. Your project now should be created with the following project structure.

# Installing Entity Framework Core

* 1. In the solution explorer, right click the solution and select “**Manage NuGet Packages For Solution**”



* 1. Search and install the following NuGet packages for Entity Framework Core.



* **Microsoft.EntityFrameworkCore**

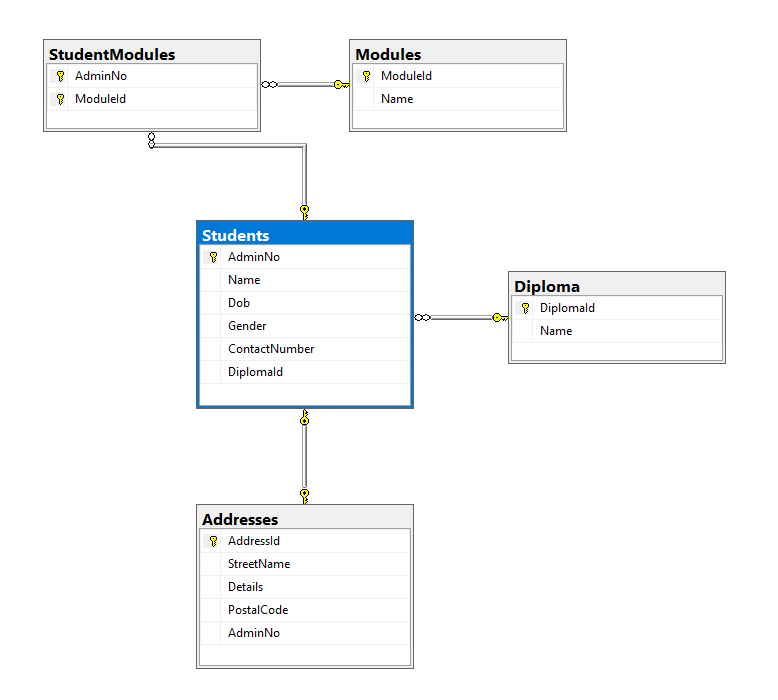
The main package that contains the core entity framework components.

* **Microsoft.EntityFrameworkCore.Tools**

Utility component add on for package manager console.

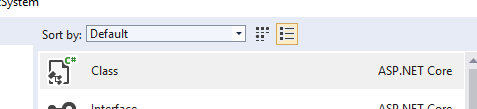
# Code First Approach:

Let’s take a look at the database schema that we will be building using entity framework core. We will be creating 4 data model and 1 join table model.



# Build Your Data Model

* + 1. In the solution explorer, right click the project/models folder and select “**Add**” > “**New Item**”. Create a new class “Student”.



* + 1. Create the properties in Student model according to the ERD. The class should have the following properties.

**Student.cs:**

|  |
| --- |
| public class Student  {  public string AdminNo { get; set; }  public string Name { get; set; }  public DateTime Dob { get; set; }  public string Gender { get; set; }  public string ContactNumber { get; set; }  } |

* + 1. Create object class for other data models.

**Address.cs:**

|  |
| --- |
| public class Address  {  public int AddressId { get; set; }  public string StreetName { get; set; }  public string Details { get; set; }  public int PostalCode { get; set; }  } |

**Diploma.cs:**

|  |
| --- |
| public class Diploma  {  public string DiplomaId { get; set; }  public string Name { get; set; }  } |

**Module.cs:**

|  |
| --- |
| public class Module  {  public string ModuleId { get; set; }  public string Name { get; set; }  } |

* + 1. Create database context. We will need to include all the models.

**StudentManagementSystemContext.cs:**

|  |
| --- |
| using Microsoft.EntityFrameworkCore;  public class StudentManagementSystemContext: DbContext  {  public StudentManagementSystemContext  (DbContextOptions<StudentManagementSystemContext> options)  : base(options)  { }  public DbSet<Diploma> Diploma { get; set; }  public DbSet<Student> Students { get; set; }  public DbSet<Address> Addresses { get; set; }  public DbSet<Module> Modules { get; set; }  } |

# Define Relationships between Entities

* + 1. Create **one to one** relationship for student address entities.

**Student.cs:**

|  |
| --- |
| public class Student  {  :  public Address Address { get; set; }  } |

**Address.cs:**

|  |
| --- |
| public class Address  {  :  public string AdminNo { get; set; }  public Student Student { get; set; }  } |

* + 1. Define delete behaviour for one to one relationship. When a student record is been deleted, we would like our address data to be deleted as well, as such we will have to define a special delete behaviour in database context.

**StudentManagementSystemContext.cs:**

|  |
| --- |
| public class StudentManagementSystemContext: DbContext  {  :  protected override void OnModelCreating(ModelBuilder modelBuilder)  {  modelBuilder.Entity<Address>()  .HasOne(i => i.Student)  .WithOne(c => c.Address)  .IsRequired()  .OnDelete(DeleteBehavior.Cascade);  }  } |

Here we will be using on **delete cascaded** behaviour, cascaded delete is a common database feature that delete related data with linked foreign key.

* + 1. Create **one to many** relationship for student diploma entities.

**Student.cs:**

|  |
| --- |
| public class Student  {  :  public string DiplomaId { get; set; }  public Diploma Diploma { get; set; }  } |

**Diploma.cs:**

|  |
| --- |
| public class Diploma  {  :  public ICollection<Student> Students { get; set; }  } |

* + 1. Create **many to many** relationship for student modules. Firstly create a new join table class “**StudentModules**” in model folder.

**StudentModules.cs:**

|  |
| --- |
| public class StudentModules  {  public string AdminNo { get; set; }  public Student Student { get; set; }  public string ModuleId { get; set; }  public Module Module { get; set; }  } |

**Modules.cs:**

|  |
| --- |
| public class Module  {  :  public ICollection<StudentModules> StudentModules { get; set; }  } |

**Student.cs:**

|  |
| --- |
| public class Student  {  :  public ICollection<StudentModules> StudentModules { get; set; }  } |

**StudentManagementSystemContext.cs:**

|  |
| --- |
| public class StudentManagementSystemContext: DbContext  {  :  public DbSet<StudentModules> StudentModules { get; set; }  protected override void OnModelCreating(ModelBuilder modelBuilder)  {  :  modelBuilder.Entity<StudentModules>()  .HasKey(sm => new { sm.AdminNo, sm.ModuleId });  modelBuilder.Entity<StudentModules>()  .HasOne(s => s.Student)  .WithMany(sm => sm.StudentModules)  .HasForeignKey(s => s.AdminNo);  modelBuilder.Entity<StudentModules>()  .HasOne(m => m.Module)  .WithMany(sm => sm.StudentModules)  .HasForeignKey(m => m.ModuleId);  }  } |

As **composite key** of join table in **StudentModules.cs** cannot be created automatically by Entity framework Core using model defined method, we will need to define the relationship manually in database context using fluent API.

# Define Constrains Using Data Annotation

Although Entity Framework Core will automatically create default constrains such as primary key, foreign key for the entities, it is not enough for real world application, thus, we will need to use data annotation to define more advance constrains.

* + 1. Constrains for students.

**Student.cs:**

|  |
| --- |
| using System.ComponentModel.DataAnnotations;  public class Student  {  //Since our primary key does not end with "ID"  [Key]  [StringLength(7)]  public string AdminNo { get; set; }  [Required]  public string Name { get; set; }  [Required]  //We only want to store date  [DataType(DataType.Date)]  //Display name  [Display(Name = "Date of Birth")]  public DateTime Dob { get; set; }  //Since we only want "M" or "F"  [Required]  [StringLength(1)]  public string Gender { get; set; }  [Required]  // A regular expression for Singapore telephone number  [RegularExpression(@"(6|8|9)\d{7}",  ErrorMessage = "Invalid Phone Number!")]  [Display(Name = "Handphone")]  public string ContactNumber { get; set; }  [Required]  [Display(Name = "Diploma")]  public string DiplomaId { get; set; }  :  } |

Other than defining keys, null or allow null (required field), we can also define input related constrains such as string length, string pattern through regular expression.

[Display()] notation is a display attribute for the properties, it will be handy in situations like Dob property where we want it to display as “Date of Birth” in the front-end instead of Dob.

* + 1. Constrains for Address.

**Address.cs:**

|  |
| --- |
| using System.ComponentModel.DataAnnotations;  using System.ComponentModel.DataAnnotations.Schema;  public class Address  {  public int AddressId { get; set; }  [Required]  public string StreetName { get; set; }  public string Details { get; set; }  [Required]  // Valid Singapore postal code range  [Range(010000, 999999)]  public int PostalCode { get; set; }  // We do not want to map this to a table field  [NotMapped]  public string FullAddress  => $"{StreetName} {Details}, Singapore {PostalCode}";    :  } |

[Not Mapped] here will indicate to entity framework core not to create a field in the database for this particular properties as it is just a handy getter method for our address data.

# Data Seeding

Data seeding is the process of inserting data when a new database table is created. This is useful for use to insert test data for development purposes.

* + 1. Insert Demo data for diploma table and course module table.

**StudentManagementSystemContext.cs:**

|  |
| --- |
| public class StudentManagementSystemContext : DbContext  {  :  protected override void OnModelCreating(ModelBuilder modelBuilder)  {  :  modelBuilder.Entity<Diploma>().HasData(  new Diploma {  DiplomaId = "C36",  Name = "Common ICT Programme" },  new Diploma {  DiplomaId = "C35",  Name = "Business & Financial Technology" },  new Diploma {  DiplomaId = "C43",  Name = "Business & Financial Technology" },  new Diploma {  DiplomaId = "C54",  Name = "Cybersecurity & Digital Forensics " },  new Diploma {  DiplomaId = "C80",  Name = "Infocomm & Security " },  new Diploma {  DiplomaId = "C85",  Name = "Information Technology " }  );  modelBuilder.Entity<Module>().HasData(  new Module {  ModuleId = "IT1010",  Name = "Data Analysis & Visualisation" },  new Module {  ModuleId = "IT1020",  Name = "Fundamentals of Innovation & Enterprise" },  new Module {  ModuleId = "IT1030",  Name = "Infocomm Security" },  new Module {  ModuleId = "IT1040",  Name = "Network Technology" },  new Module {  ModuleId = "IT1050",  Name = "Web Development" },  new Module {  ModuleId = "IT1060",  Name = "Programming Essentials" },  new Module {  ModuleId = "ITX150",  Name = "App Development Project" }  );  }  } |

# Generate the Database

We will use the CLI tools provided by the Entity Framework tools Nuget extension to generate our database using our data models.

* + 1. Configure the database connection.

To make **StudentManagementSystemContext** available to MVC controllers, register it as a service in **Startup.cs**.

As a best practice and security reasons, we will put our database connection string in a configuration file.

**Appsettings.json:**

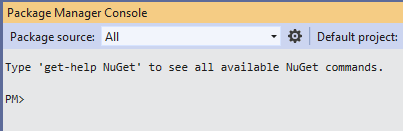
|  |
| --- |
| {  :  "ConnectionStrings": {  "StudentManagementSystemDatabase": "Server=(localdb)\\EfCore;Database=StudentManagementSystem;Trusted\_Connection=True;"  }  } |

**Startup.cs:**

|  |
| --- |
| using Microsoft.EntityFrameworkCore;  using StudentManagementSystem\_CodeFirst.Models;  public class Startup  {  public void ConfigureServices(IServiceCollection services)  {  :  var connectionString =  Configuration  .GetConnectionString("StudentManagementSystemDatabase");  services.AddDbContext<StudentManagementSystemContext>  (options => options.UseSqlServer(connectionString));  }  } |

* + 1. Create the database.

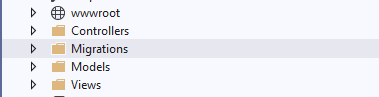
Open Package Manager Console: **Tools > NuGet Package Manager > Package Manager Console.**



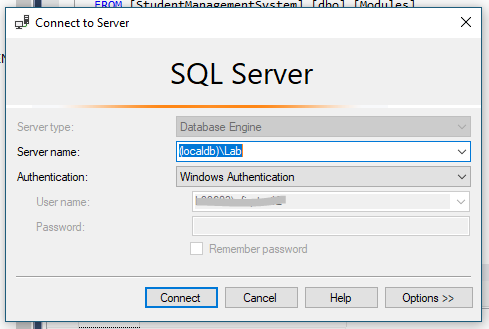
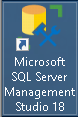
Run the following commands:

|  |
| --- |
| Add-Migration InitialCreate  Update-Database |

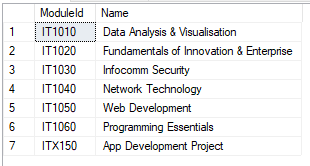
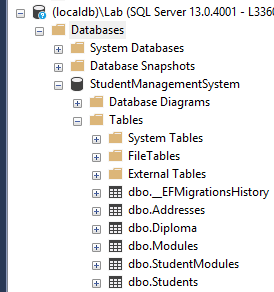
The following file will be created automatically.



* + 1. You can view the database created using **SQL Server Management Studio**. Use the server name “**(localdb)\Lab**” and windows authentication, click “**Connect**” to launch the studio.



You should now be able to view the database created by Entity Framework Core.

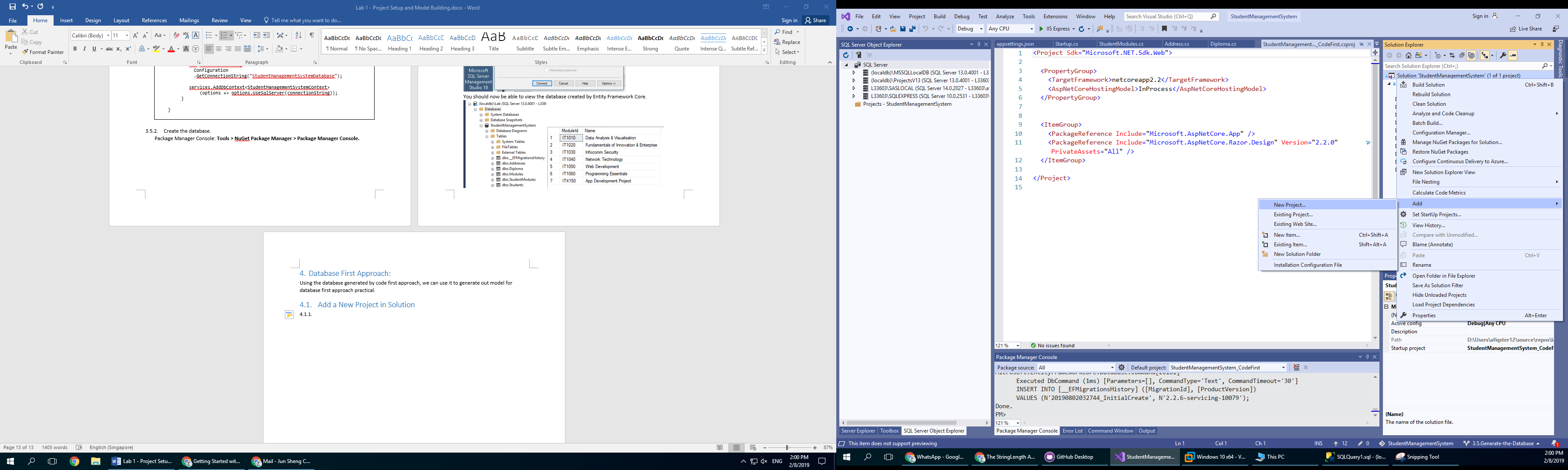


# Database First Approach:

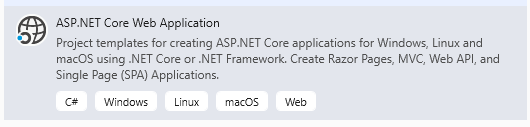
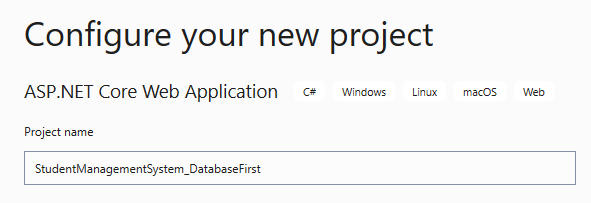
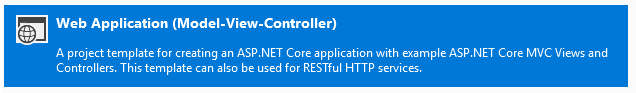
Using the database generated by code first approach, we can use it to generate data models for database first approach practical.

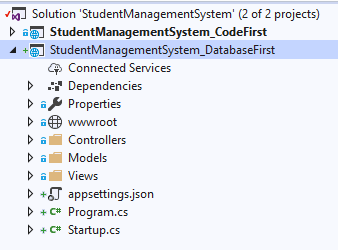
# Add a New Project in Solution

* + 1. Right click the solution in solution explorer, select “**Add**” > “**New Project…**”

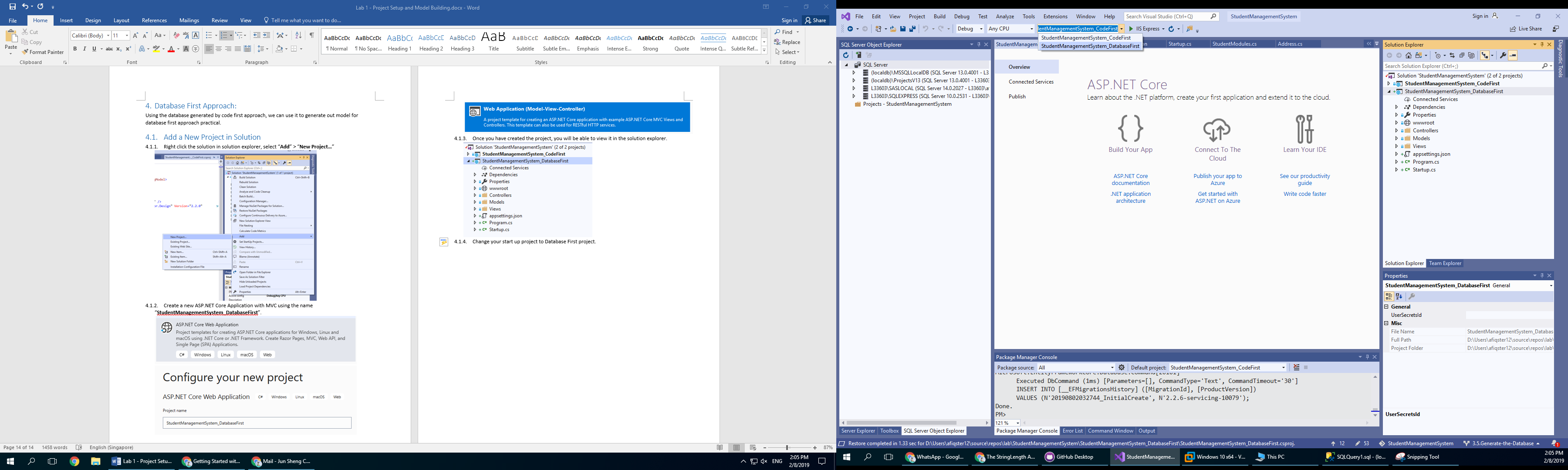


* + 1. Create a new ASP.NET Core Application with MVC using the name “**StudentManagementSystem\_DatabaseFirst**”.

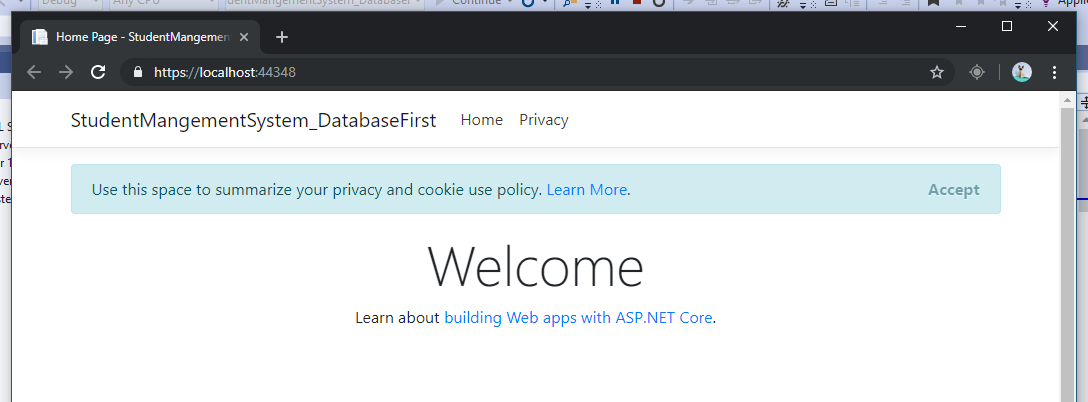
1. 
2. 
3. 
   * 1. Once you have created the project, you will be able to view it in the solution explorer.



* + 1. Change your start up project to Database First project.



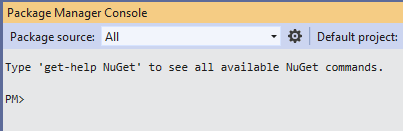
You should now be able to launch your web application.



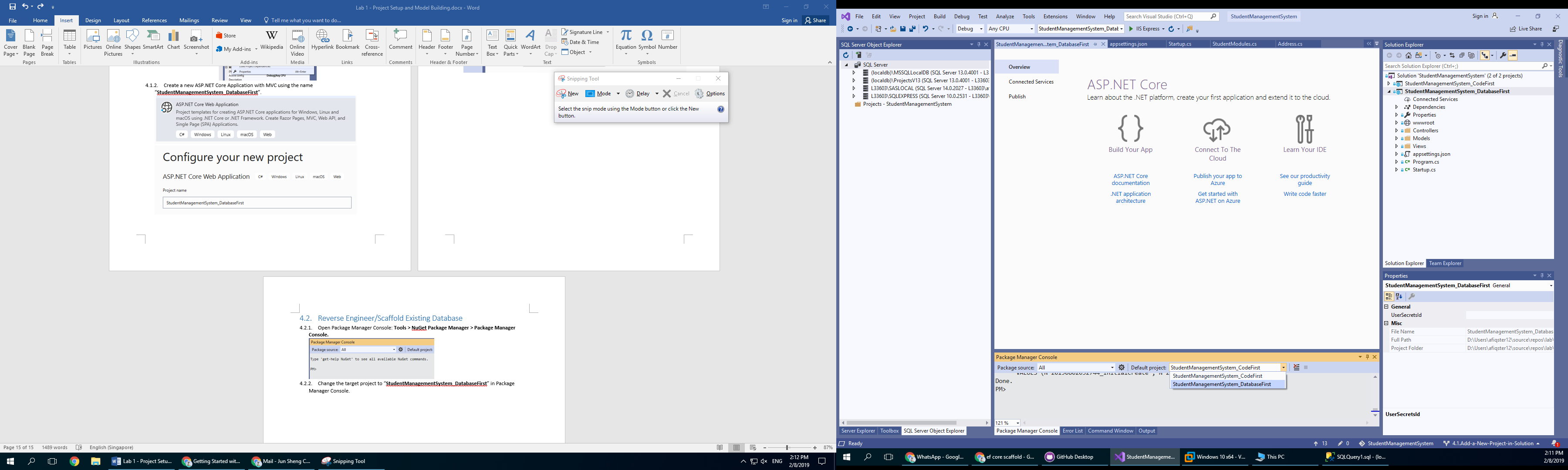
# Reverse Engineer/Scaffold Existing Database

To generate our data model using existing database, we will need to execute a cli command in package manager. The command we will be using is “**Scaffold-DbContext**” which is already installed through NuGet. Supplied with our database connection string, it will automatically scaffold our database schema and generated out data model.

* + 1. Open Package Manager Console: **Tools > NuGet Package Manager > Package Manager Console.**

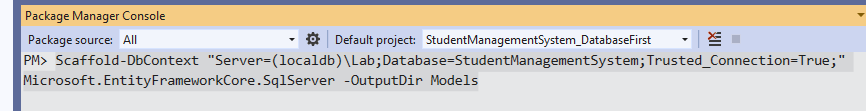


* + 1. Change the target project to “**StudentManagementSystem\_DatabaseFirst**” in Package Manager Console such that it is targeting the correct output directory.



* + 1. Run the following command.

|  |
| --- |
| Scaffold-DbContext "Server=(localdb)\Lab;Database=StudentManagementSystem;Trusted\_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models |



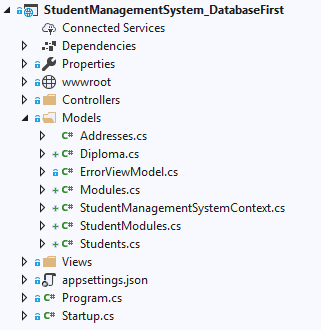
Supply the correct connection string to Scaffold-DbContext command.

(You can get the connection string from **StudentManagementSystem\_CodeFirst** project **> appsettings.json**)

**Take Note:**

|  |  |
| --- | --- |
|  | Ensure No double **back slash** in the database connection string. |

* + 1. After the command completed successfully, you will be able to find all generated data models in the models folder.



# Take a look at the Generated Files

As you can see, most of the generated code is similar to what we have created using the code first approach. The only difference is that the entity relationship is defined in database context using fluent API instead of model defined relationship mapping.



**Take Note:**

As our database is relatively simple and does not have many complex relationship mapping involved, model defined relationship mapping method is easier for us to implement. However, if our database schema involves many complex relationships, fluent API relationship mapping will be more effective to work with.

No matter which approach, both methods will provide you with working data object interface to interact with the database.