Entity Framework Core

Tan Cher Wah (cherwah@nus.edu.sg)

Entity Framework Core

- Entity Framework Core is an Object Relational Mapper (ORM) that **maps Models** (C# classes) in our application to the **columns and tables** of a relational database

- The developer is offered a **higher level of abstraction** when manipulating data - instead of thinking of columns, rows and tables, he now thinks in terms of his Models

Entity Framework Core

- For example, to read from the database, the developer simply **performs queries against his Models**
- And to update the database, the developer simply **updates his Models**
- All these without having the developer issues explicit SQL calls

Database First

- In a Database First approach, a database designer first designs a **database schema** to define required tables, columns, relationships, indexes and constraints

- Then, the developer develops application code based on the provided database schema

Code First

- In a Code First approach (for Entity Framework Core), the developer writes C# **code to define and create** tables, columns, relationships, indexes and constraints for the application's database

- Hence, instead of using a database tool to create our database, we use code

Data Types

When using EF Core, C# data-types are automatically converted to SQL data-types

C#	SQL
int	int
long	bigint
float	float
string	nvarchar
DateTime	datetime
Guid	UniqueIdentifier
bool	bit

MODELS

Model

- In EF Core, a **Model** (C# class) is mapped to a database **table**
- A database **column** is represented as an **automatic property** in the Model
- Any property named **Id** shall be the **Primary Key**

```
public class ModelName
{
    public ModelName() {
        Id = new Guid();
    }

    /* maps to primary key */
    public Guid Id { get; set; }

    /* maps to N table-columns */
    public <DATA_TYPE> <PROPERTY_NAME>  { get; set; }

    ...
}
```

Data Annotations

Data Annotations can be specified in our Models to create database constraints; some common ones are listed below.

Data Annotation	Effect
[Required]	The column cannot be NULL
[MaxLength(<size>)	The max length for a column
[Column(<name>)	Override the default column-name
[Key]	Set as primary key (instead of Id)
[ForeignKey(<name>)	Set as foreign key

Data Annotations

Data Annotations applied to a property in our earlier Model

```
public class ModelName
{
    public <ModelName>()
    {
        Id = new Guid();
    }

    /* maps to primary key */
    public Guid Id { get; set; }

    /* property data annotation */
    [Required]
    [MaxLength(<SIZE>)]
    [Column(<PREFERRED_COLUMN_NAME>)]
    public <DATA_TYPE> <PROPERTY_NAME>  { get; set; }

    ...
}
```

Data Annotations

Navigational Properties

- Navigational Properties can be used to define **table-relationships** on our Models
- Navigational Properties also allow easy navigation from one Model to another in a **query** (i.e. simulates a **Join** operation)

Navigational Property

A Navigational Property is always preceded by the keyword 'virtual'

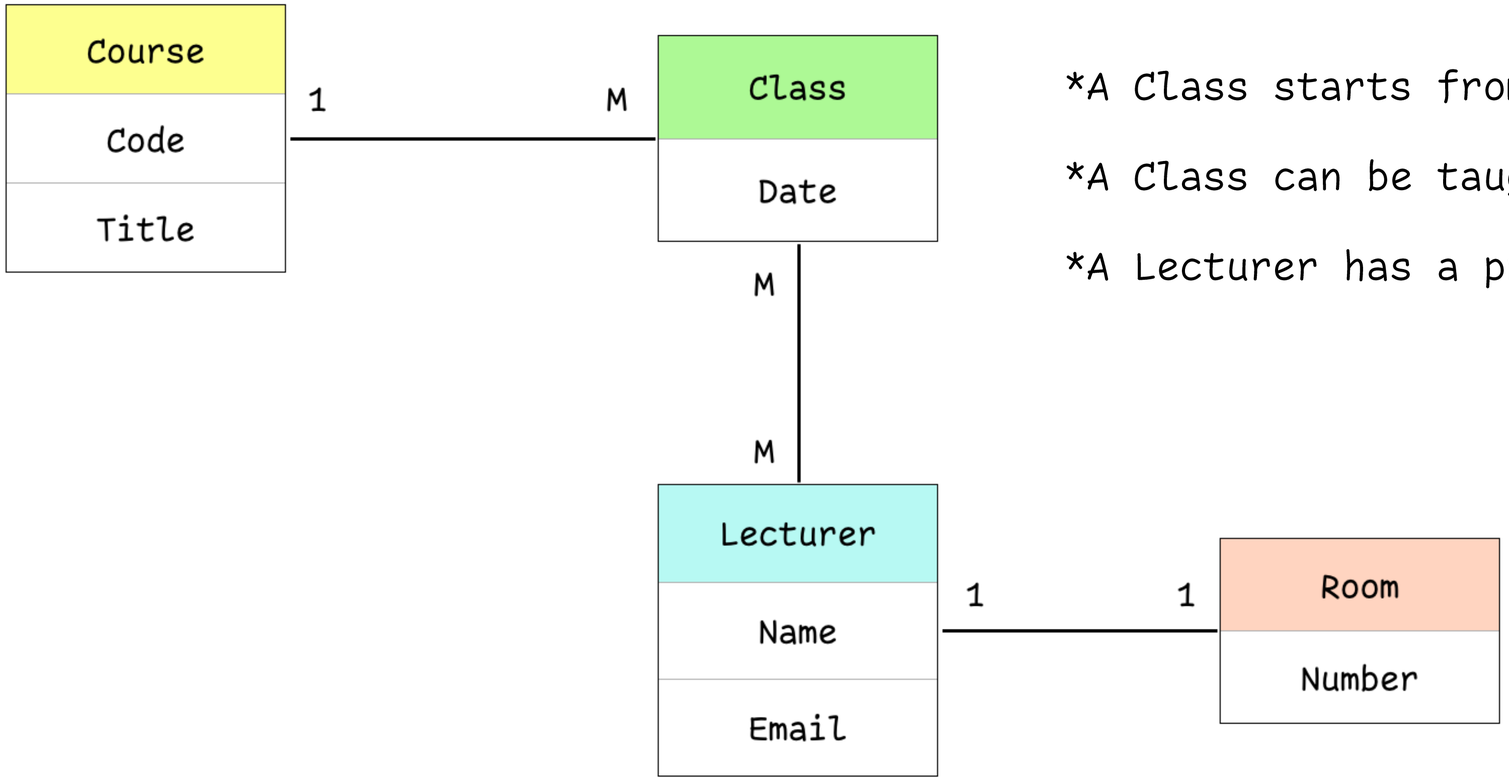
```
public class ModelName
{
    public <ModelName>()
    {
        Id = new Guid();
    }

    /* maps to primary key */
    public Guid Id { get; set; }

    /* property with data annotation */
    [Required]
    [MaxLength(<SIZE>)]
    [Column(<PREFERRED_COLUMN_NAME>)]
    public <DATA_TYPE> <PROPERTY_NAME>  { get; set; }

    /* navigation properties - notice the virtual keyword */
    public virtual <DATA_TYPE> <PROPERTY_NAME> { get; set; }

    . . .
}
```

Navigational Properties

TABLE RELATIONSHIPS

Sample Entities

How can we use the Code First approach to create a database to represent the following tables and relationships?



*A Class starts from 9am to 5pm

*A Class can be taught by more than one Lecturer

*A Lecturer has a private Room

Course model

A bare-bone Course class to model a Course database table

```
public class Course
{
    public Course()
    {
        Id = new Guid();
    }

    public Guid Id { get; set; }
    public string Code { get; set; }
    public string Title { get; set; }
}
```

Class model

A bare-bone Class class to model a Class database table

```
public class Class
{
    public Class()
    {
        Id = new Guid();
    }

    public Guid Id { get; set; }
    public DateTime Date { get; set; }
}
```

Lecturer model

A bare-bone Lecturer class to model a Lecturer database table

```
public class Lecturer
{
    public Lecturer()
    {
        Id = new Guid();
    }

    public Guid Id { get; set; }
    public string Name { get; set;}
    public string Email { get; set; }
}
```

Room model

A bare-bone Room class to model a Room database table

```
public class Room
{
    public Room()
    {
        Id = new Guid();
    }

    public Guid Id { get; set; }
    public string Number { get; set; }
}
```

One-to-One Relationship

- To model a one-to-one relationship between Lecturer and Room, have a Navigational Property of Room in Lecturer and ... (next slide)

```
public class Lecturer
{
    public Lecturer()
    {
        Id = new Guid();
        Class = new List<Class>();
    }

    public Guid Id { get; set; }
    public string Name { get; set;}
    public string Email { get; set; }

    public virtual Room Room { get; set; }

    public virtual ICollection<Class> Class { get; set; }
}
```

One-to-One Relationship

- ... and a Navigation Property of Lecturer in Room
- EF Core will create a LecturerId column (Foreign Key constraint) in table Room

```
public class Room
{
    public Room()
    {
        Id = new Guid();
    }

    public Guid Id { get; set; }
    public string Number { get; set; }

    [ForeignKey("LecturerId")]
    public virtual Lecturer Lecturer { get; set; }

}
```

One-to-Many Relationship

- To model a one-to-many relationship between Course and Class, specify a Navigational Property for a collection of Class in Course and ... (next slide)

```
public class Course
{
    public Course()
    {
        Id = new Guid();
        Class = new List<Class>();
    }

    public Guid Id { get; set; }
    public string Code { get; set; }
    public string Title { get; set; }

    public virtual ICollection<Class> Class { get; set; }

}
```

One-to-Many Relationship

- ... and a Navigational Property for a Course in Class

```
public class Class
{
    public Class()
    {
        Id = new Guid();
        Lecturer = new List<Lecturer>();
    }

    public Guid Id { get; set; }
    public DateTime Date { get; set; }

    public virtual Course Course { get; set; }

    public virtual ICollection<Lecturer> Lecturer { get; set; }
}
```

Many-to-Many Relationship

- To model a Many-to-Many relationship between Class and Lecturer, specify a Navigation Property to hold a collection of Lecturer in Class and ... (next slide)

```
public class Class
{
    public Class()
    {
        Id = new Guid();
        Lecturer = new List<Lecturer>();
    }

    public Guid Id { get; set; }
    public DateTime Date { get; set; }

    public virtual Course Course { get; set; }

    public virtual ICollection<Lecturer> Lecturer { get; set; }
}
```

Many-to-Many Relationship

- ... and a Navigational Property to hold a collection of Class in Lecturer
- EF Core will then create a new table, named ClassLecturer, with only columns ClassId and LecturerId

```
public class Lecturer
{
    public Lecturer()
    {
        Id = new Guid();
        Class = new List<Class>();
    }

    public Guid Id { get; set; }
    public string Name { get; set;}
    public string Email { get; set; }

    public virtual Room Room { get; set; }

    public virtual ICollection<Class> Class { get; set; }
}
```

DATABASE CONTEXT

DbContext and DbSet

- DbContext represents our Database
- DBSet represents a Database Table in our database

```
public class MyDbContext : DbContext
{
    public MyDbContext(DbContextOptions<MyDbContext> options)
        : base(options)
    {
        // this empty constructor is needed because we need
        // to pass the options to the parent class
    }

    public DbSet<Course> Course { get; set; }
    public DbSet<Class> Class { get; set; }
    public DbSet<Lecturer> Lecturer { get; set; }
    public DbSet<Room> Room { get; set; }
}
```

Database Connection String

In appsettings.json, store credentials for connecting to our database server (MSSQL)

```
{
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft.AspNetCore": "Warning"
        }
    },
    "AllowedHosts": "*",
    "ConnectionStrings": {
        "conn_str":
"Server=localhost\\SQLEXPRESS;Database=<database_name>;TrustServerCertificate=True
;Trusted_Connection=True"
    }

}
```

Add Database Context as Dependency

In Program.cs, add our Database Context as a dependency

```
...

// Add services to the container.
builder.Services.AddControllersWithViews();

// Add our database context (using AddScoped() under the hood).
builder.Services.AddDbContext<MyDbContext>(options => {
    var conn_str = builder.Configuration.GetConnectionString("conn_str");
    options.UseLazyLoadingProxies().UseSqlServer(conn_str);
});

var app = builder.Build();

...
```

Create our Database

In Program.cs, use our Database Context to create a database

```
...

// Initialize our database before our web application starts
InitDB(app.Services);

app.Run();

void InitDB(IServiceProvider serviceProvider)
{
    using var scope = serviceProvider.CreateScope();
    MyDbContext db = scope.ServiceProvider.GetRequiredService<MyDbContext>();

    // for our debugging, we just start off by removing our old
    // database (if there is one).
    db.Database.EnsureDeleted();

    // create a new database.
    db.Database.EnsureCreated();
}
```

DATA MANIPULATION

Dependency Injection

Our Database Context can be injected into our Controllers as a dependency

```
public class HomeController : Controller
{
    private readonly MyDbContext db;

    public HomeController(MyDbContext db)
    {
        this.db = db;
    }

...
```

Dependency Injection happens here

Add

- Use Add() to create new rows in both tables - Lecturer and Room
- SaveChanges() must be called for changes to take effect
- The following adds a new row in both table Room and Lecturer

```
public IActionResult AddDemo()
{
    Lecturer lecturer = new Lecturer {
        Name = "Michael Jordan"
    };

    db.Add(new Room {
        Number = "05-10",
        Lecturer = lecturer
    });

    db.SaveChanges();

...
```

FirstOrDefault

- Use FirstOrDefault() to look for the first entry that matches a search criteria
- If there are no matches, a NULL is returned

```
public IActionResult FirstOrDefaultDemo()
{
    Room? room = db.Room.FirstOrDefault(x =>
        x.Lecturer.Name == "Michael Jordan"
    );

    if (room != null)
    {
        Debug.WriteLine("{0} is in room {1}",
            room.Lecturer.Name, room.Number);
    }

...
```

Where

- Use Where() to return a list of models that matches a search criteria
- If there are no matches, the returned List is empty

```
public IActionResult WhereDemo()
{
    List<Class> classes = db.Class.Where(x =>
        x.Lecturer.Count() > 1 && x.Course.Code == "NET"
    ).ToList();

    foreach (Class c1 in classes)
    {
        List<Lecturer> lecturers = (List<Lecturer>) c1.Lecturer.ToList();

        Debug.WriteLine("{0} is taught by {1} and {2}",
            c1.Course.Title, lecturers[0].Name, lecturers[1].Name);
    }

...
```

Remove

- To remove a record in the database, first query against relevant Model
- Then remove the returned model(s)

```
public IActionResult RemoveDemo()
{
    Lecturer? lecturer = db.Lecturer.FirstOrDefault(x =>
        x.Name == "Michael Jordan"
    );

    if (lecturer != null)
    {
        db.Remove(lecturer);
        db.SaveChanges();
    }

...
```

Update

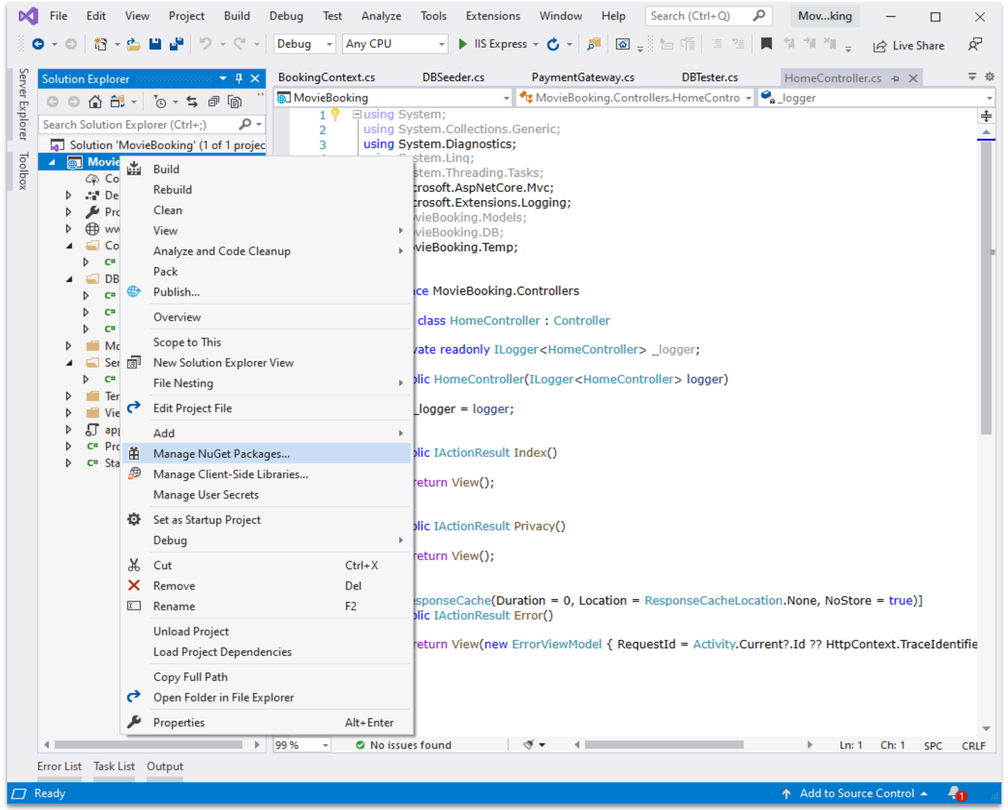
- To update a record in the database, first query against relevant Model
- Then make changes to the returned model(s)

```
public IActionResult UpdateDemo()
{
    Lecturer? lecturer = db.Lecturer.FirstOrDefault(x =>
        x.Name == "Michael Jordan"
    );

    if (lecturer != null)
    {
        lecturer.Room.Number = "04-11";
        db.SaveChanges();
    }

...
```

SETTING UP TO USE ENTITY FRAMEWORK CORE

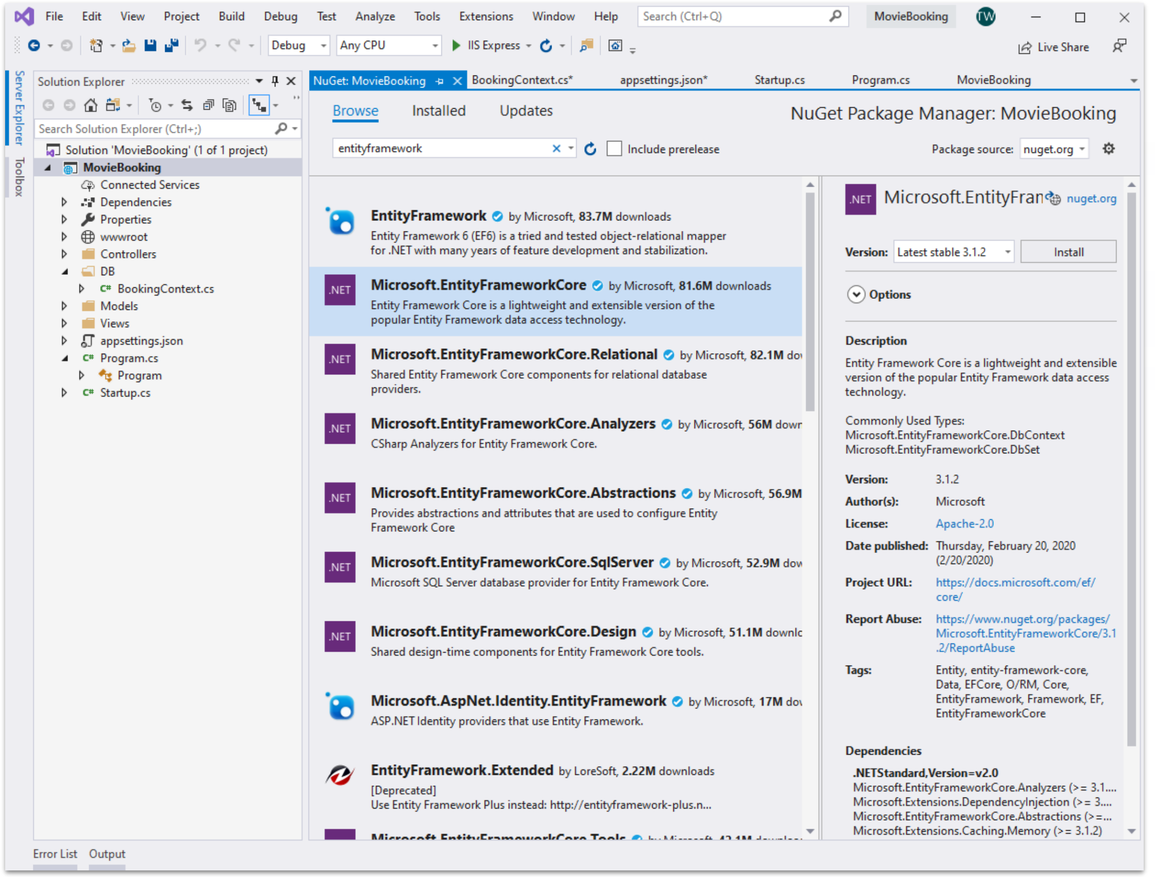
1 – Locate NuGet Package Manager

- NuGet is a Package Manager for .NET, used for distributing software written using the .NET framework
- Select “Manage NuGet Packages…”



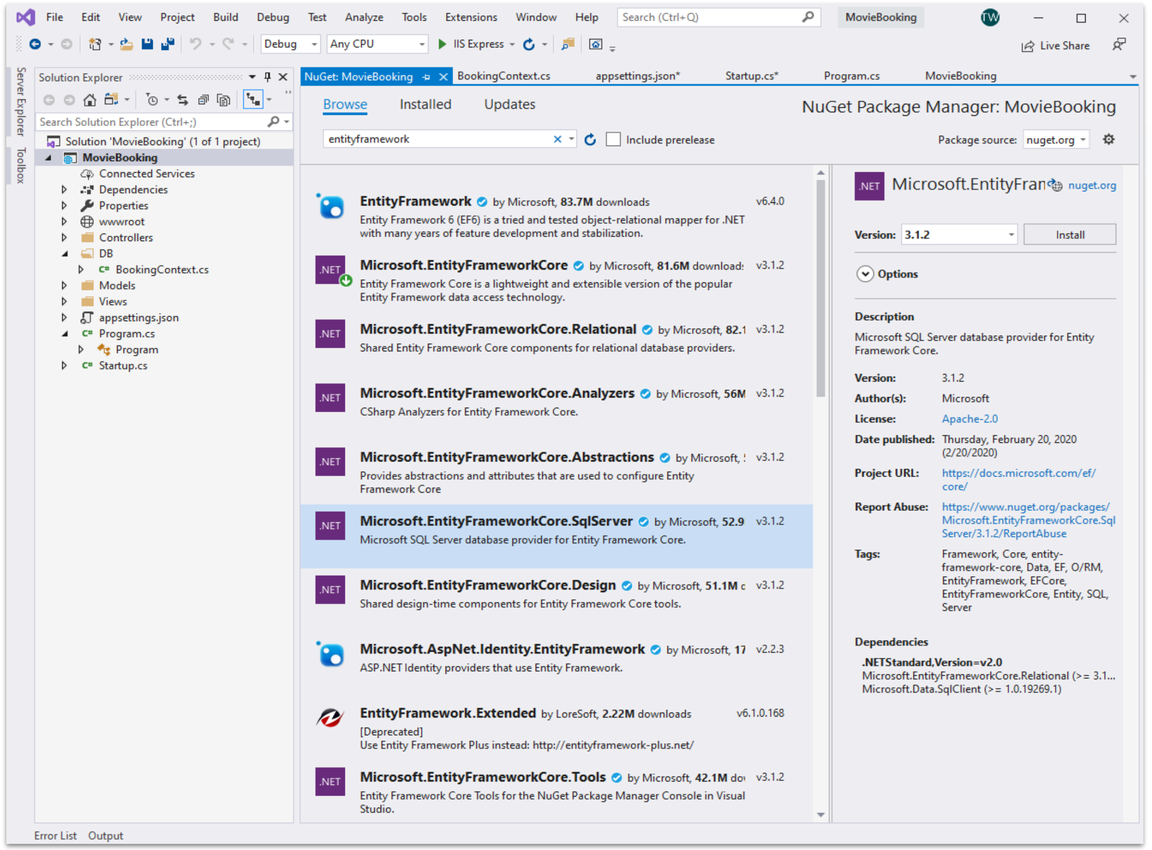
2 – Install Entity Framework Core

- Search for and install Microsoft.EntityFrameworkCore
- Microsoft's ORM for .NET Core



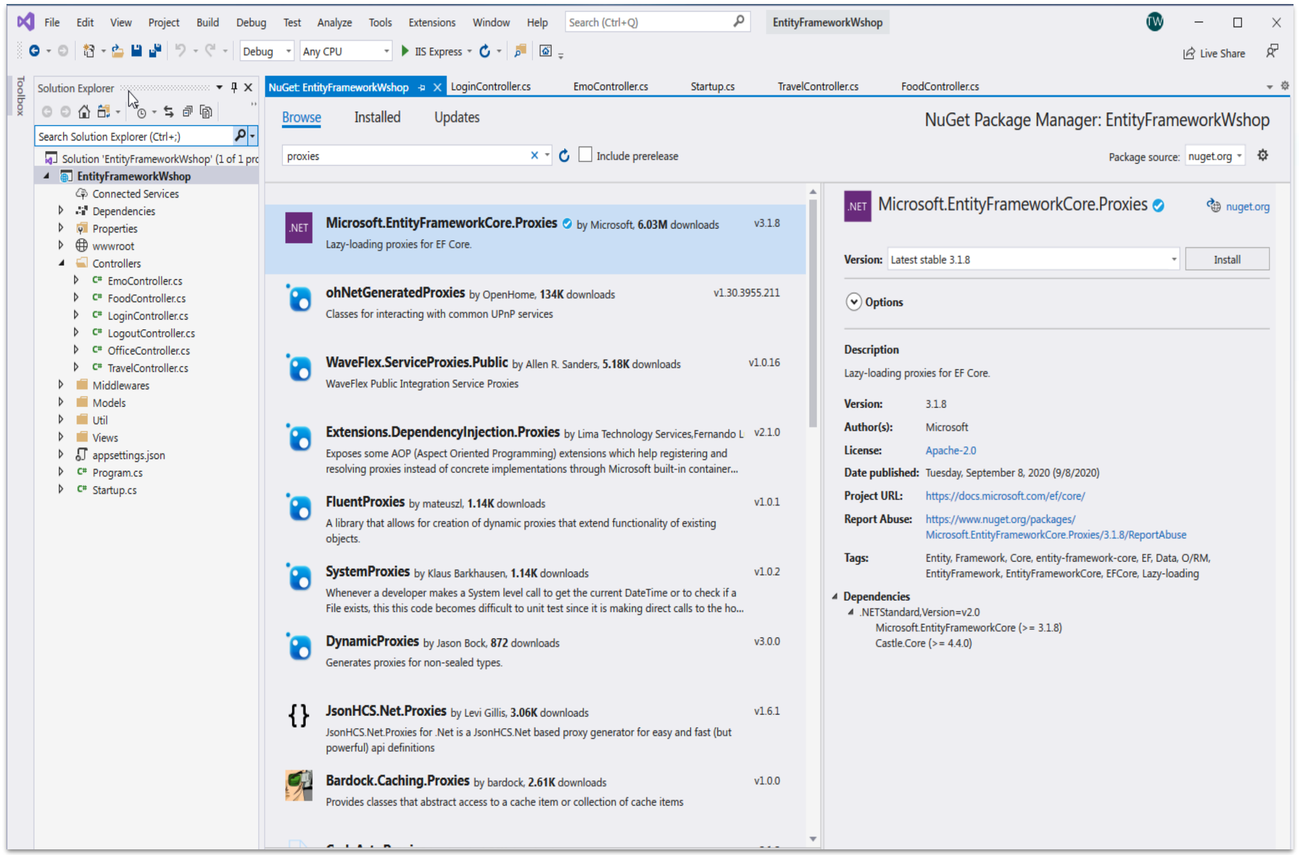
3 - Install Data Provider

- Search and Install Microsoft.EntityFrameworkCore.SqlServer
- It facilitates ORM against a Microsoft SQL Server



4 – Install Proxy

- Search and Install Microsoft.EntityFrameworkCore.Proxies
- It simplifies loading of data your navigation properties point to



THE END