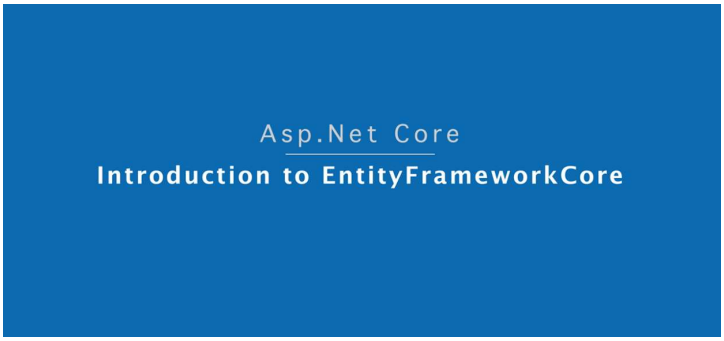
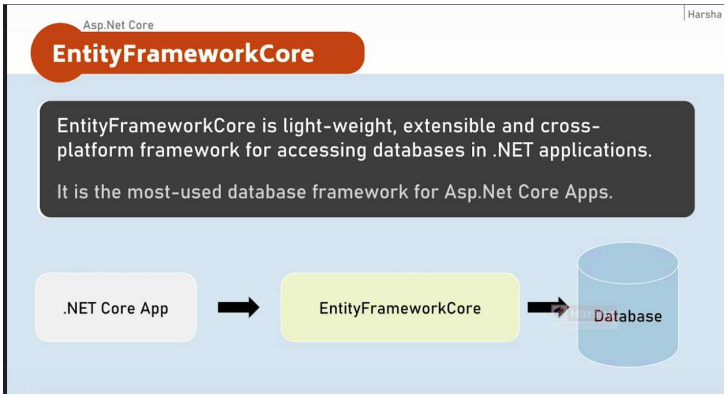
Section 18: EntityFrameworkCore [MVC and Web API]

Tuesday, February 4, 2025 10:05 AM

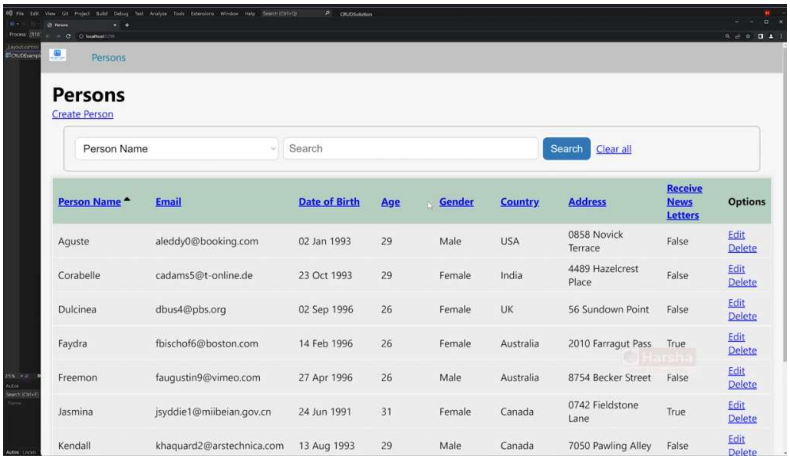


Entity Framework Core

It is one of the most commonly used database frameworks with ASP.NET Core applications.
It is a lightweight, extensible, cross-platform framework for connecting to databases as a part of .NET Core applications.
So why do you require database connections in your projects?



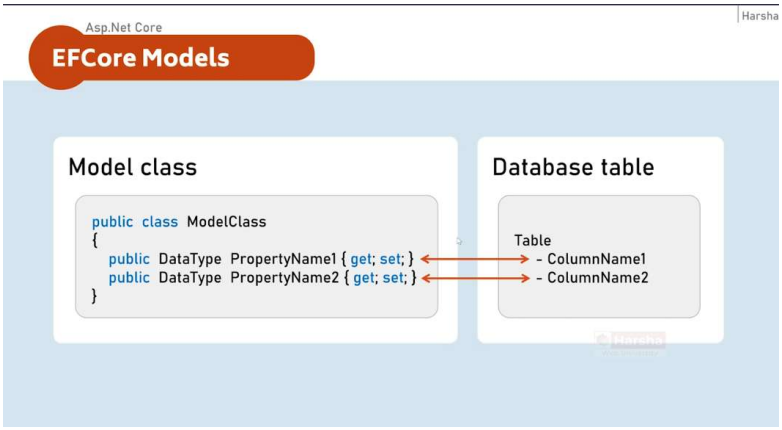
See, what about the information that you are maintaining currently? For example, a person's information or countries—all of that is temporary.

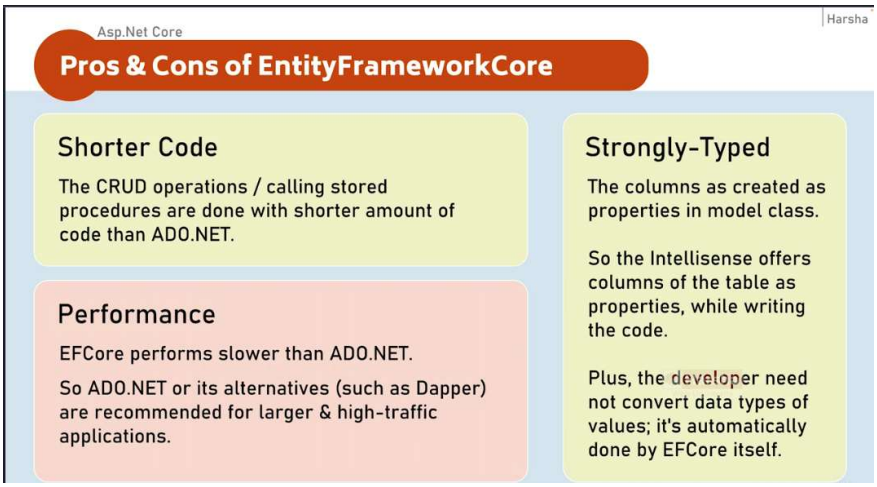


For example, if you make any changes here—let's say I add the digit "1" after the name—it seems to be updated.
But next time, when you close and restart the application, the changes are gone.
Why? Because the changes you make are stored in a temporary in-memory collection, which gets destroyed as soon as the application closes.

To overcome this problem, we need to store data permanently, and databases are the solution.
Every real-world application requires a database, and it also needs a database framework to connect with the database from the code.
Entity Framework Core is the tool or framework that helps you connect with databases efficiently.

Particularly in Entity Framework Core, you need to keep your model classes in sync with your actual database tables.
In general, a table contains columns, and the model class is coupled with the table, where each property name is mapped to the corresponding column.
By convention, property names should be the same as column names, but it is not mandatory.
You may use different names if required and map them to specific columns using data annotations.
However, following the practice of keeping property names the same as column names is a common convention.
so you have to create one model class
for one table
in your database if you have 30 tables
for example you have to create 30 model
classes





Benefits of Using Entity Framework Core Compared to Traditional ADO.NET
When comparing Entity Framework Core with traditional database connections such as ADO.NET, there are several benefits:

1. **Less Code, Faster Development**
   - With Entity Framework Core, you write a shorter amount of code.
   - You can quickly create a DbContext object and write LINQ queries against it to perform query operations.
   - Calling SaveChanges() automatically handles updates, insertions, or deletions based on changes made to entities.

2. **Simplified Code & Object Management**
   - In ADO.NET, you need to create multiple objects like SqlConnection, SqlDataAdapter, SqlCommand, SqlDataReader, and more.
   - Entity Framework Core abstracts this complexity, allowing you to focus on business logic rather than database interactions.

3. **Strongly Typed Data**
   - In ADO.NET, column names are written as string literals (e.g., dataReader["PersonName"].ToString()), which is error-prone and requires typecasting.
   - In Entity Framework Core, properties are strongly typed, meaning values are automatically typecasted based on the declared type (e.g., Person.Name as a string).
   - This reduces errors

Benefits of Using Entity Framework Core Compared to Traditional ADO.NET
When comparing Entity Framework Core with traditional database connections such as ADO.NET, there are several benefits:

1. **Less Code, Faster Development**
   - With Entity Framework Core, you write a shorter amount of code.
   - You can quickly create a DbContext object and write LINQ queries against it to perform query operations.
   - Calling SaveChanges() automatically handles updates, insertions, or deletions based on changes made to entities.
2. **Simplified Code & Object Management**
   - In ADO.NET, you need to create multiple objects like SqlConnection, SqlDataAdapter, SqlCommand, SqlDataReader, and more.
   - Entity Framework Core abstracts this complexity, allowing you to focus on business logic rather than database interactions.
3. **Strongly Typed Data**
   - In ADO.NET, column names are written as string literals (e.g., dataReader["PersonName"].ToString()), which is error-prone and requires typecasting.
   - In Entity Framework Core, properties are strongly typed, meaning values are automatically typecasted based on the declared type (e.g., Person.Name as a string).
   - This reduces errors and eliminates the need for manual type conversion.
4. **Automatic Mapping of Columns to Properties**
   - Instead of manually handling column-to-property mapping, Entity Framework Core automatically maps database columns to class properties.
   - If you have 20 columns in a table, you define 20 properties in your entity, and values are directly available in those properties.

Drawback of Entity Framework Core

- **Slower Performance Compared to ADO.NET**
  - Entity Framework Core is built on top of ADO.NET, meaning it introduces an abstraction layer.
  - Even though it is highly optimized, it cannot outperform raw ADO.NET queries.
  - For **data-intensive applications** with high traffic, direct ADO.NET queries or alternatives like **Dapper** may be preferable.

When to Use Entity Framework Core?

- **Recommended for small to medium applications** due to ease of development.
- **Good for prototyping** in the early stages of development.
- **Not ideal for high-performance applications** handling large amounts of data.

What's Next?

In this section, we will explore Entity Framework Core and perform CRUD operations using LINQ queries as well as stored procedures.

-
  - and eliminates the need for manual type conversion.
4. **Automatic Mapping of Columns to Properties**
   - Instead of manually handling column-to-property mapping, Entity Framework Core automatically maps database columns to class properties.
   - If you have 20 columns in a table, you define 20 properties in your entity, and values are directly available in those properties.

Drawback of Entity Framework Core

- **Slower Performance Compared to ADO.NET**
  - Entity Framework Core is built on top of ADO.NET, meaning it introduces an abstraction layer.
  - Even though it is highly optimized, it cannot outperform raw ADO.NET queries.
  - For **data-intensive applications** with high traffic, direct ADO.NET queries or alternatives like **Dapper** may be preferable.

When to Use Entity Framework Core?

- **Recommended for small to medium applications** due to ease of development.
- **Good for prototyping** in the early stages of development.
- **Not ideal for high-performance applications** handling large amounts of data.

What's Next?

In this section, we will explore Entity Framework Core and perform CRUD operations using LINQ queries as well as stored procedures.

Two Approaches in Entity Framework Core: Database First vs. Code First

Entity Framework Core can be used in **two approaches**, also known as **two ways**:

1. **Database First Approach**
2. **Code First Approach**

Difference Between Database First & Code First Approach

- **Database First Approach**
  - You **generate entity model classes** based on an **existing database**.
  - The database structure is already defined, and Entity Framework extracts models from it.

- **Code First Approach**
  - You **create entity model classes first**, and the **database is generated automatically** using migrations.
  - You apply migration commands (dotnet ef migrations add and dotnet ef database update) to reflect changes in the database.

Handling Database Changes

Aspect	Database First Approach	Code First Approach
Modifying database schema	Changes must be made directly in the database using SQL commands.	Changes are made in entity model classes and applied using migrations.
Syncing models with the database	After modifying the database, models must be manually updated to match.	Entity Framework automatically syncs models with the database via migrations.
Control over database	More control over database design, making it suitable for	Easier for new projects where database structure

structure	legacy or large-scale databases.	evolves with development.

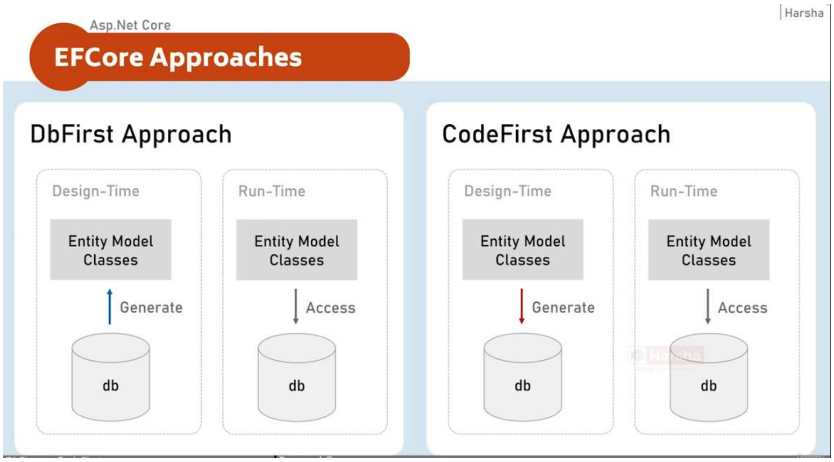
Which One to Use?

☑ **Choose Database First If:**
- You are working with an **existing database**.
- You need **precise control** over the database schema.
- The database is **managed by DBAs** separately from application code.

☑ **Choose Code First If:**
- You are **starting a new project** from scratch.
- You prefer to manage the database **directly from code**.
- You want **automatic database migrations** instead of manual SQL changes.

Runtime Behavior (Same for Both Approaches)
- Queries written using **LINQ** will retrieve data from the database.
- Calling **SaveChanges()** will apply **insert, update, and delete** operations to the database.
- The runtime model remains the same regardless of the approach used.

Conclusion
- **Database First** is better for existing databases and projects with strict DB control.
- **Code First** is better for new projects where database structure evolves with development.
- **Both approaches work the same at runtime**, so the choice depends on **how you prefer to manage database schema changes**.



Which Approach is Better? When to Prefer Which One?
Both **Code First** and **Database First** approaches have their own advantages and use cases. The choice depends on factors like project requirements, database management preferences, and long-term scalability.

Code First Approach: Best for New Applications

☑ **When to Use Code First:**
- When developing a **new application** from scratch.
- When you **don't have an existing database**.
- When you prefer to manage the database schema **directly from C# models**.
- When you want to use **automatic database migrations** to evolve the schema.
- When working in **small teams** where the database structure changes frequently.
- For **prototyping and proof-of-concept applications** before client approval.

❌ **Limitations of Code First:**
- **Limited database control**—DBA teams may not prefer this approach.
- Direct SQL modifications are **discouraged**, as migrations should handle schema changes.
- **Less flexibility** when dealing with database-specific optimizations like indexes, stored procedures, triggers, and complex queries.
- Risk of **losing manual database changes** when migrating to another environment.

Database First Approach: Best for Existing Databases
☑ **When to Use Database First:**
- When **working with an existing database** (migrating an old application).
- When the **DBA team manages the database independently**.
- When the database requires **manual optimizations** like indexes, stored procedures, and triggers.
- When performance is a **high priority** and you need **precise control** over database schema.
- When dealing with **large-scale, data-intensive applications** with high traffic.

❌ **Limitations of Database First:**

- Requires **manual synchronization** between database schema and entity models.
- **More effort** when making schema changes (developers must update both the database and code manually).
- **Less flexibility for rapid development**, as database modifications require SQL changes first.

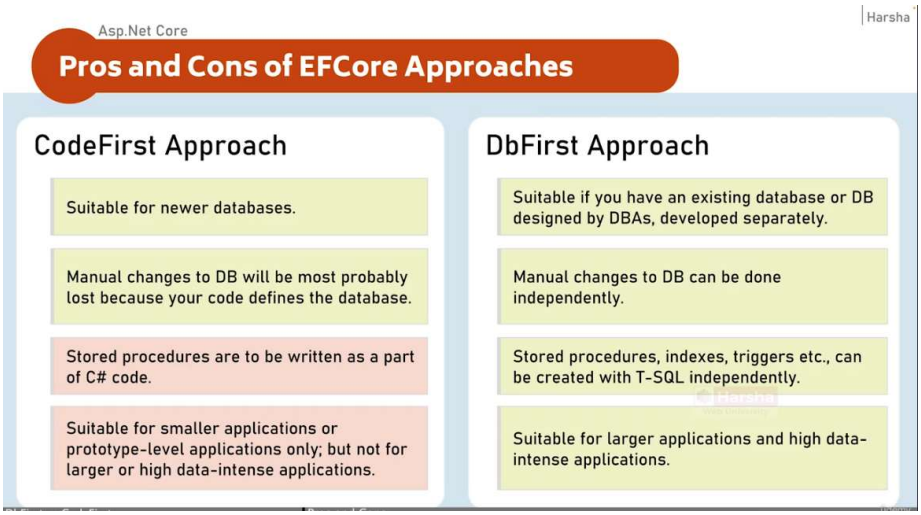
Which Approach to Choose Based on Application Size?

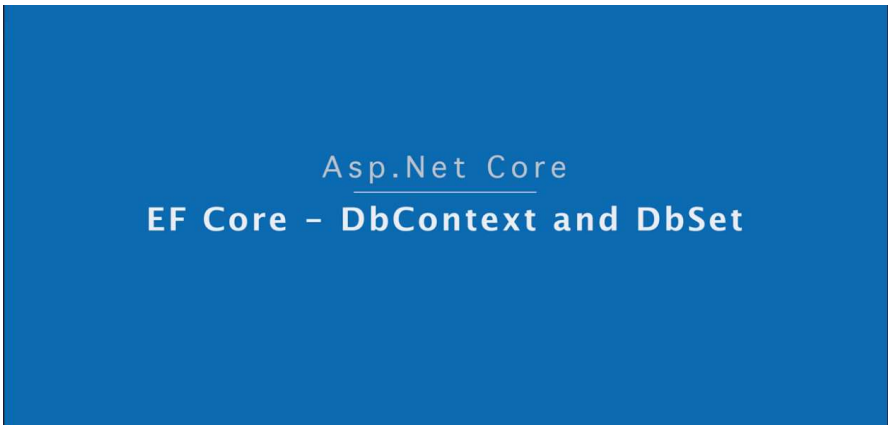
Application Size	Recommended Approach
Small Applications / Prototypes	Code First
New Mid-Sized Applications	Code First or Database First
Large Applications with Heavy Data Usage	Database First
High-Traffic Enterprise Systems	Database First or Custom ORM (e.g., Dapper)

Can You Switch Between Code First and Database First?
Yes! Many developers start with **Code First** for prototyping and then **migrate to Database First** as the project grows and requires more database control.

Final Recommendation
- **For learning purposes, start with Code First** since it simplifies development.
- **For real-world projects, consider Database First** if the database needs independent management.
- **For high-performance applications, consider Database First or an alternative like Dapper.**

In the next sections, we will explore **Entity Framework Core using the Code First approach**, as it is the best starting point for learning EF Core. 🚀



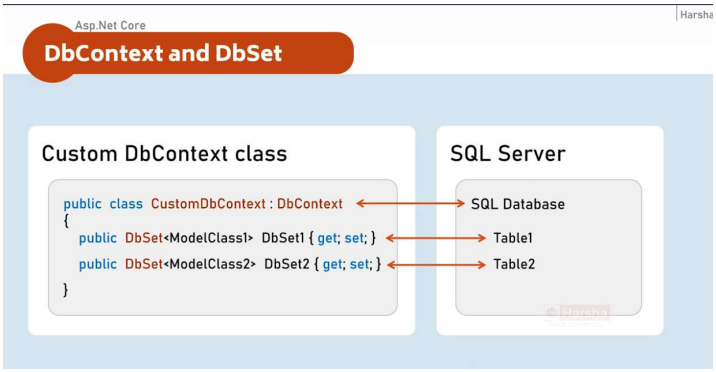


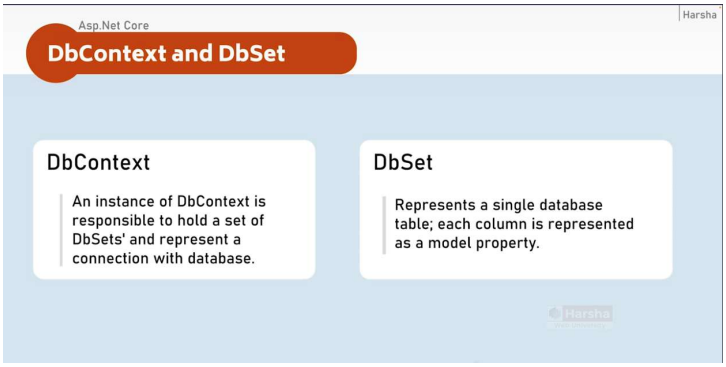
In order to begin with Entity Framework Core, you require to know DbContext and DbSet. The DbContext is bound to a specific SQL database, and DbSet is bound to a specific table.

For example, in your SQL Server, you have a database with two tables. So, for the database, you will be creating one DbContext, and for each table, you will be creating one DbSet.
The DbSet is a generic class, so you require to pass the generic parameter, and that is your model class.
For each table, you will be creating a model class, right? That model class will be mentioned here.

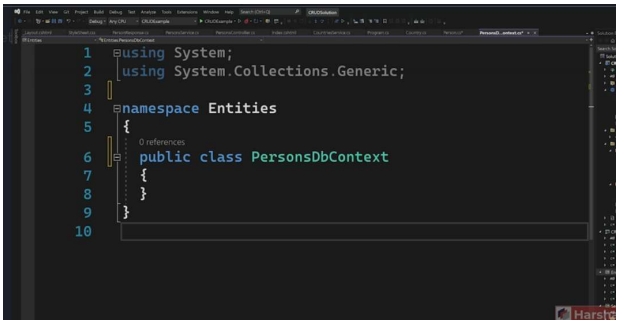
For example, we want two tables like Countries and Persons, and the model class names will be singular, that is Country and Person, respectively. So, you will be creating DbSet<Country> and DbSet<Person>.
It is a convention to name the DbSet as plural, meaning the DbSet names will be Countries and Persons, respectively.
Here, your DbContext class name can be anything, but it must be inherited from a predefined class called DbContext from the Microsoft.EntityFrameworkCore namespace.

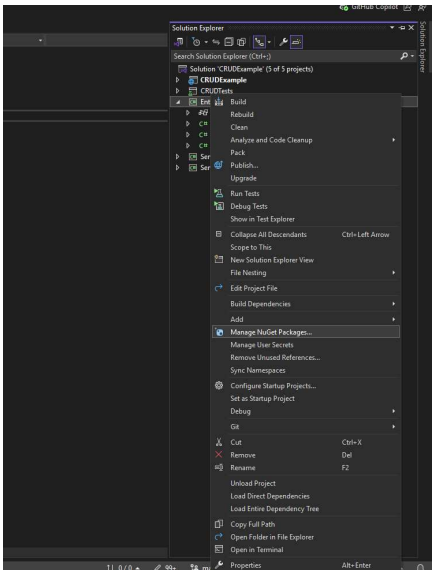
In the same way, in the same namespace, you have the DbSet class as well.
So, in short, the DbContext represents the entire database, but DbSet represents a single table or maybe a database view.
OK, let me demonstrate the same practically.

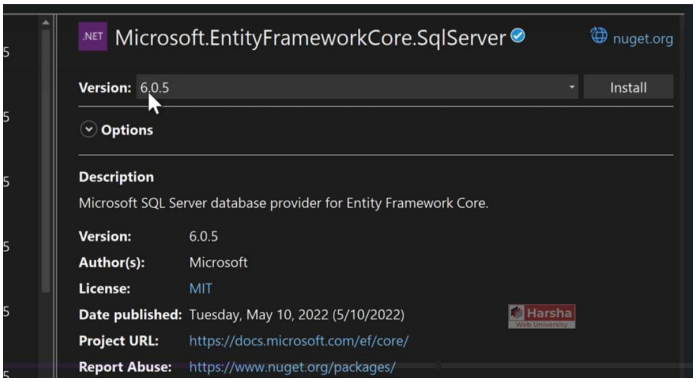




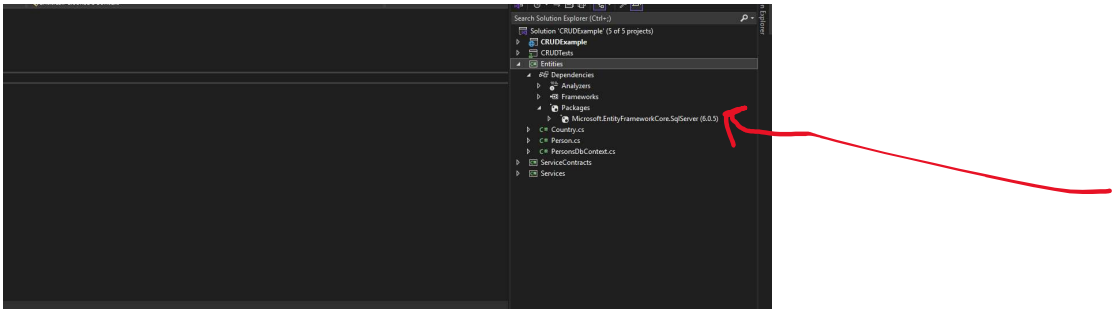
But it is a normal class, we have to convert it into a DbContext class, To do that, we need to download a nuget package.



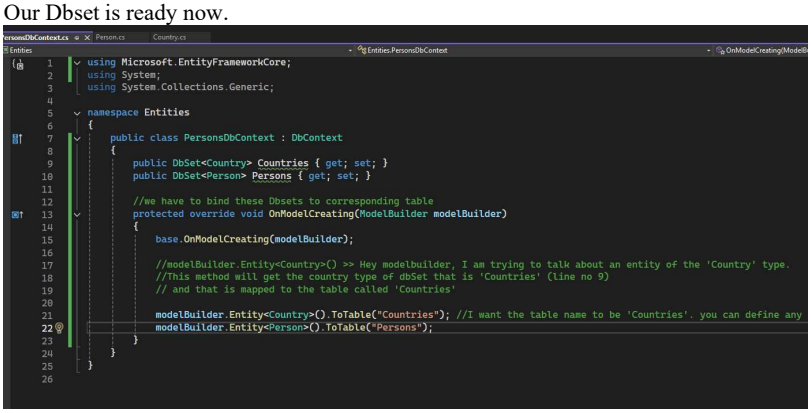




To ensure that package is installed or not.



Our Dbset is ready now.

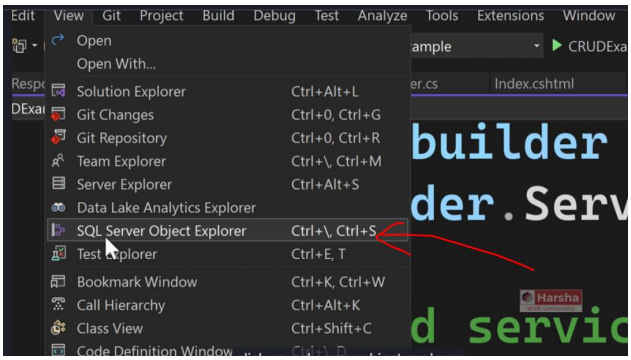


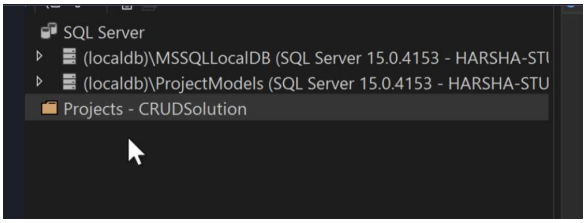
```
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;

namespace Entities
{
    public class PersonsDbContext : DbContext
    {
        public DbSet<Country> Countries { get; set; }
        public DbSet<Person> Persons { get; set; }

        //we have to bind these Dbsets to corresponding table
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);

            //modelBuilder.Entity<Country>() >> Hey modelbuilder, I am trying to talk about an entity of the 'Country' type.
            //This method will get the country type of dbSet that is 'Countries' (line no 9)
            // and that is mapped to the table called 'Countries'

            modelBuilder.Entity<Country>().ToTable("Countries"); //I want the table name to be 'Countries'. you can define any
            modelBuilder.Entity<Person>().ToTable("Persons");
        }
    }
}
```

After that, we have to add our db set as a service in your program.cs file. Because EF core by default uses 'Dependency Injection'

A connection string is a string value that specifies the details of the database you want to connect to.
In the initial stages of development, it is recommended to use the local database. This means your SQL Server database file will be created as part of the same developer machine.
If there are multiple developers, each developer would have their own database on their own machine, ensuring no conflicts between databases of multiple developers.

In order to create a new database go to the view menu, click on sql server object explorer.



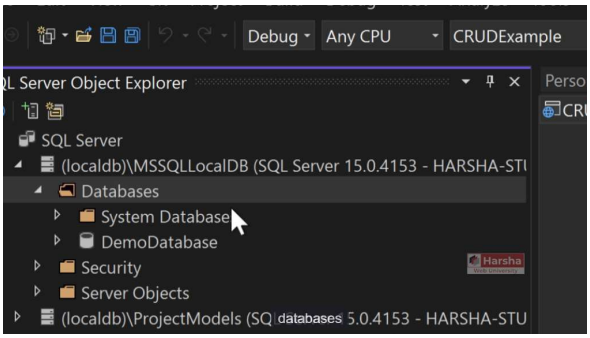


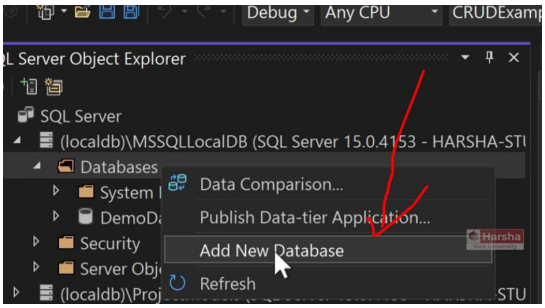
Here you can see your local DB instances.
SQL Server mainly supports two types of database instances:

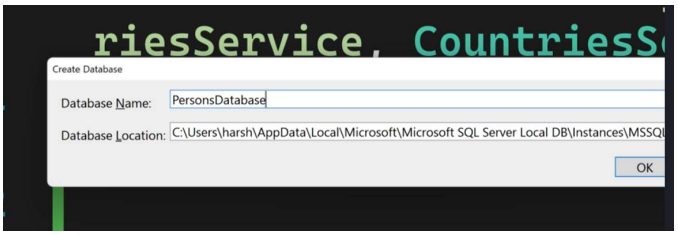
1. **Standard SQL Server database instance** – Connected based on the server name or IP address.
2. **LocalDB** – This LocalDB database file is created as part of the same developer machine.

For staging and production environments, maintaining a dedicated centralized database server is recommended. However, for development scenarios, it is recommended to use LocalDB.

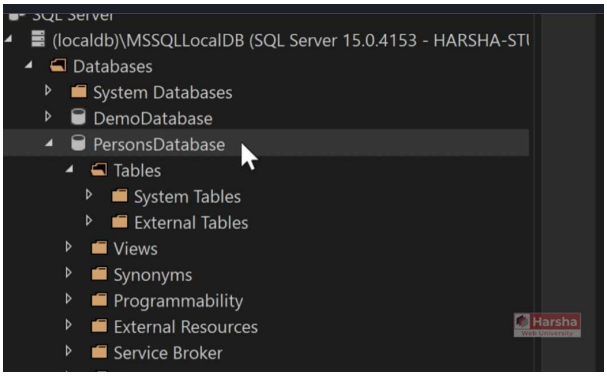
When you install Visual Studio, it installs LocalDB by default, named **MSSQLLocalDB**, which is the name of the LocalDB server instance.
In the SQL Server Object Explorer, expand LocalDB and click on **Databases** to see the list of existing databases.
Now, I would like to create a new one.

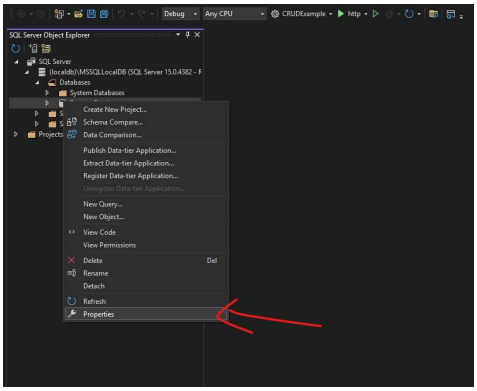




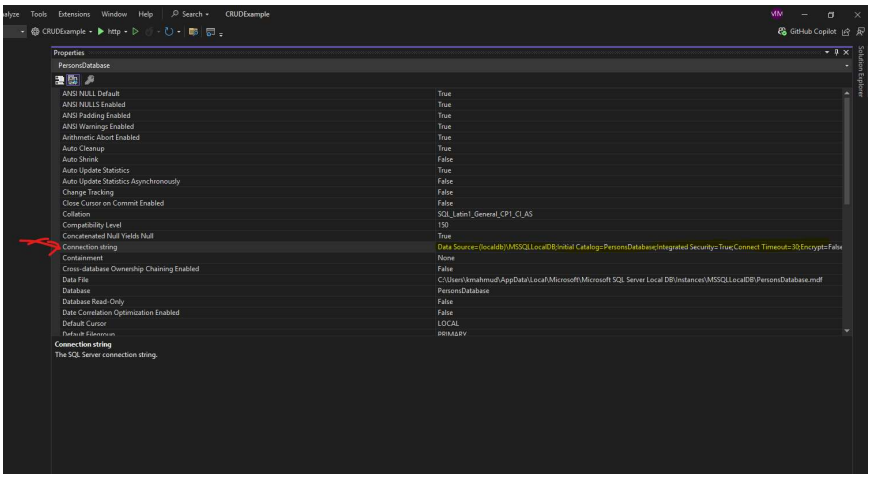


There are no table by default, We would like to connect to this database. But we have to mention the connection details as connection string.



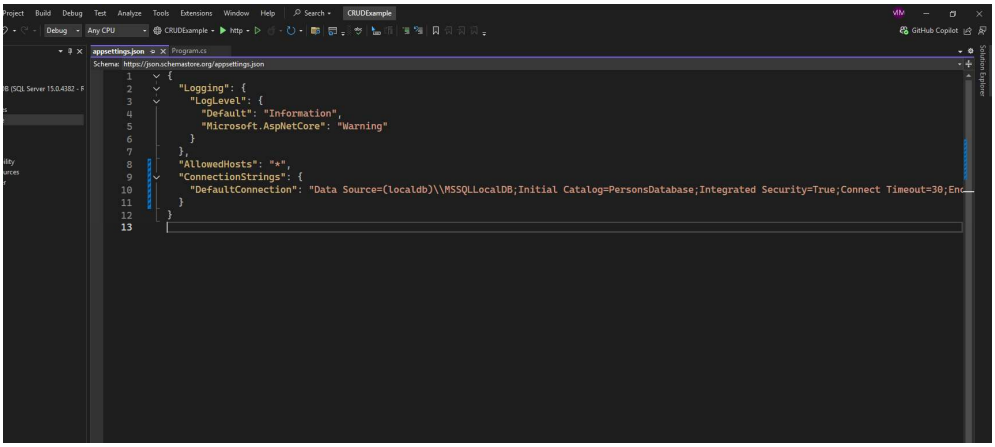


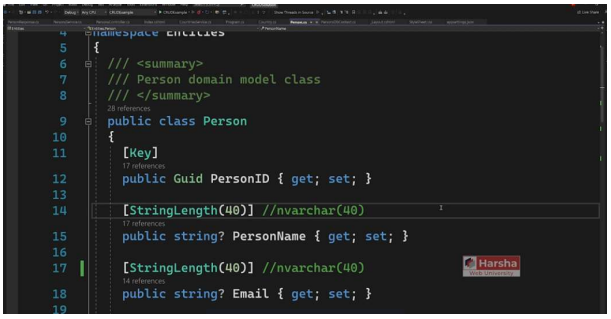
Copy this connection string.



Now paste the connection string into program.cs file.

Open your Configuration file appSettings.json file and paste here.

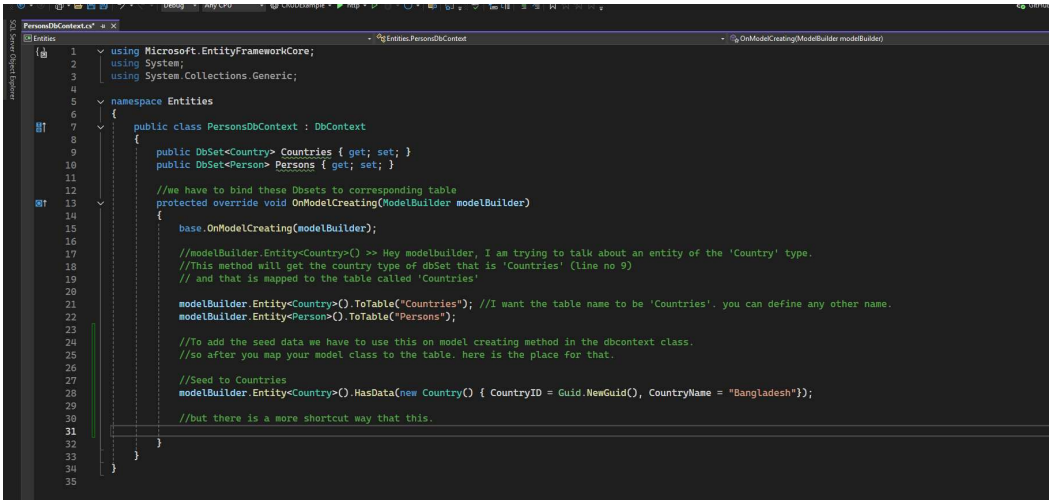




Okay, assume that we have created the Persons and Countries table, and by default, those tables will be empty, right? Assume there is a Persons table with all these columns, but by default, it has no rows. But generally, we would like to add some initial data into the table, and that is called seed data here.

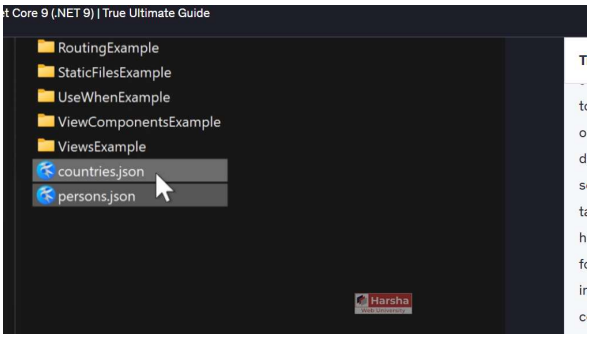
How to add seed data in Entity Framework?

We have been provided with a method called HasData. If you supply the entity object to it, that object (i.e., that row) will be inserted into the database table as soon as the database is created initially.
However, it will not be reinserted if you delete it, meaning this data will not be maintained in the table if you make any changes to the table. These are the initial rows that get added only when the table is created, not after that.
So, let me add the seed data.

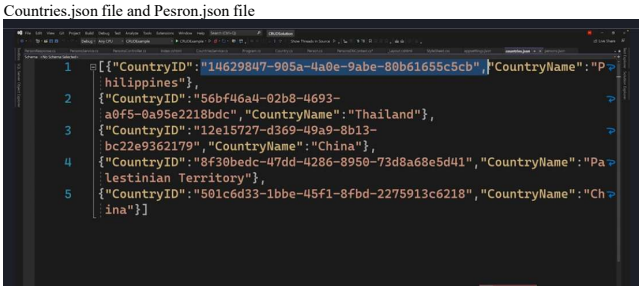


```
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;

namespace Entities
{
    public class PersonsDbContext : DbContext
    {
        public DbSet<Country> Countries { get; set; }
        public DbSet<Person> Persons { get; set; }

        //We have to bind these Dbsets to corresponding table
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);

            //modelBuilder.Entity<Country>() >> Hey modelbuilder, I am trying to talk about an entity of the 'Country' type.
            //This method will get the country type of dbSet that is 'Countries' (line no 9)
            // and that is mapped to the table called 'Countries'

            modelBuilder.Entity<Country>().ToTable("Countries"); //I want the table name to be 'Countries'. you can define any other name.
            modelBuilder.Entity<Person>().ToTable("Persons");

            //To add the seed data we have to use this on model creating method in the dbcontext class.
            //so after you map your model class to the table. here is the place for that.

            //Seed to Countries
            modelBuilder.Entity<Country>().HasData(new Country() { CountryID = Guid.NewGuid(), CountryName = "Bangladesh"});

            //but there is a more shortcut way that this.
        }
    }
}
```

Paste the json file into your Project folder.

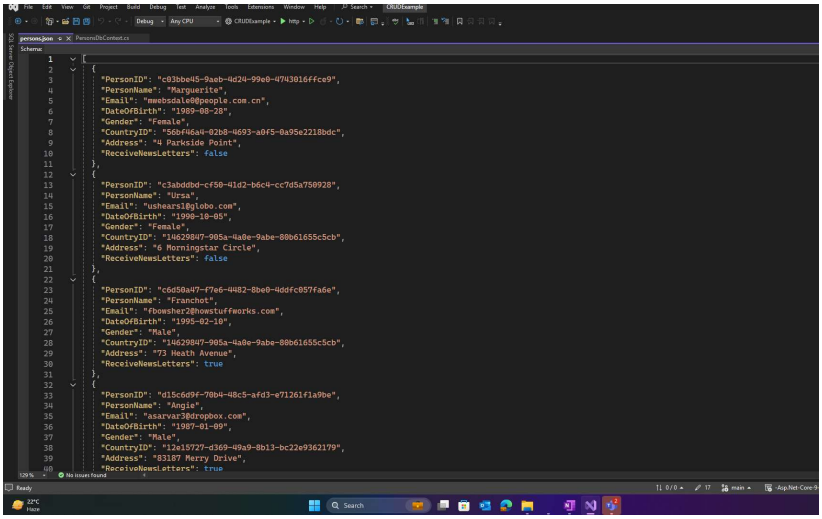


Countries.json file and Pesron.json file



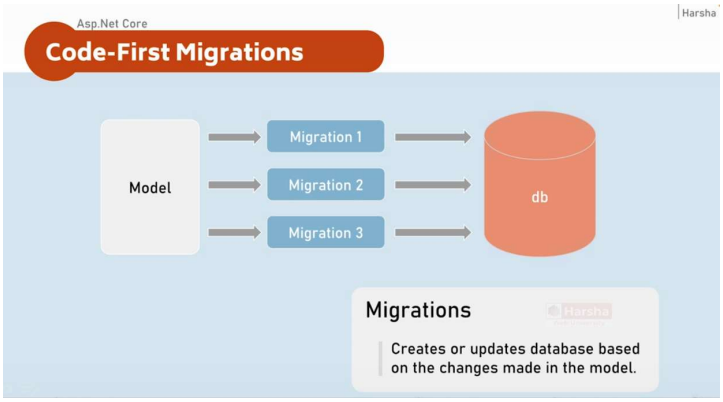
```
[{"CountryID":"14629847-905a-4a0e-9abe-80b61655c5cb","CountryName":"Philippines"},
{"CountryID":"56bf46a4-02b8-4693-a0f5-0a95e2218bdc","CountryName":"Thailand"},
{"CountryID":"12e15727-d369-49a9-8b13-bc22e9362179","CountryName":"China"},
{"CountryID":"8f30bedc-47dd-4286-8950-73d8a68e5d41","CountryName":"Palestinian Territory"},
{"CountryID":"501c6d33-1bbe-45f1-8fbd-2275913c6218","CountryName":"China"}]
```

CountryId of the Person should map to Countries.json. CountryId is a foreign key to the Person table. Assume that.



```
[
  {
    "PersonID": "c03bbe45-9aeb-4d24-99e8-47f43016ffce9",
    "PersonName": "Marguerite",
    "Email": "mmebsdale0@people.com.cn",
    "DateOfBirth": "1989-08-28",
    "Gender": "Female",
    "CountryID": "56bf46a4-02b8-4693-a0f5-0a95e2218bdc",
    "Address": "4 Parkside Point",
    "ReceiveNewsLetters": false
  },
  {
    "PersonID": "c3abddbd-cf50-41d2-b6c4-cc7d5a750928",
    "PersonName": "Ursa",
    "Email": "ushears1@glolo.com",
    "DateOfBirth": "1990-10-05",
    "Gender": "Female",
    "CountryID": "14629847-905a-4a0e-9abe-80b61655c5cb",
    "Address": "6 Morningstar Circle",
    "ReceiveNewsLetters": false
  },
  {
    "PersonID": "c6d50a47-f7e6-4482-8be8-4ddfc057fa6e",
    "PersonName": "Franchot",
    "Email": "fbowsher2@howstuffworks.com",
    "DateOfBirth": "1995-02-10",
    "Gender": "Male",
    "CountryID": "14629847-905a-4a0e-9abe-80b61655c5cb",
    "Address": "73 Heath Avenue",
    "ReceiveNewsLetters": true
  },
  {
    "PersonID": "d15c6d9f-70b4-48c5-afd3-e71261f1a9be",
    "PersonName": "Angie",
    "Email": "asarvar3@dropbox.com",
    "DateOfBirth": "1987-01-09",
    "Gender": "Male",
    "CountryID": "12e15727-d369-49a9-8b13-bc22e9362179",
    "Address": "83187 Merry Drive",
    "ReceiveNewsLetters": true
```

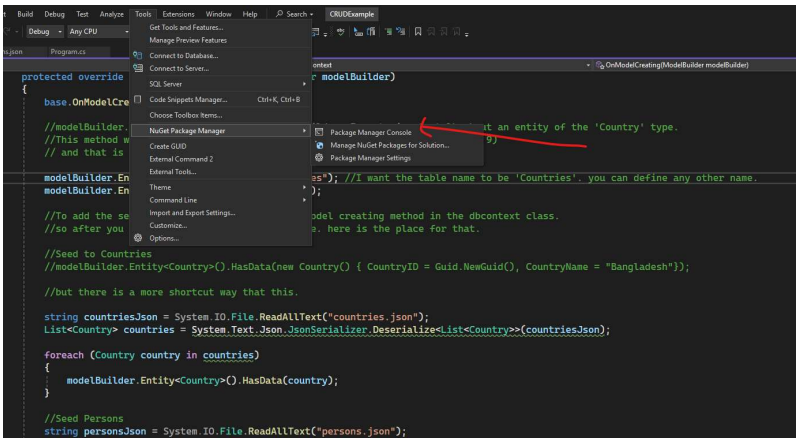
```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    //modelBuilder.Entity<Country>() >> Hey modelbuilder, I am trying to talk about an entity of the 'Country' type.
    //This method will get the country type of dbSet that is 'Countries' (line no 9)
    // and that is mapped to the table called 'Countries'

    modelBuilder.Entity<Country>().ToTable("Countries"); //I want the table name to be 'Countries'. you can define any other name
    modelBuilder.Entity<Person>().ToTable("Persons");

    //To add the seed data we have to use this on model creating method in the dbcontext class.
    //so after you map your model class to the table. here is the place for that.

    //Seed to Countries
    //modelBuilder.Entity<Country>().HasData(new Country() { CountryID = Guid.NewGuid(), CountryName = "Bangladesh"});

    //but there is a more shortcut way that this.

    string countriesJson = System.IO.File.ReadAllText("countries.json");
    List<Country> countries = System.Text.Json.JsonSerializer.Deserialize<List<Country>>(countriesJson);

    foreach (Country country in countries)
    {
        modelBuilder.Entity<Country>().HasData(country);
    }

    //Seed Persons
    string personsJson = System.IO.File.ReadAllText("persons.json");
    List<Person> persons = System.Text.Json.JsonSerializer.Deserialize<List<Person>>(personsJson);

    foreach (Person pesron in persons)
    {
        modelBuilder.Entity<Person>().HasData(pesron);
    }

}
}
}
```



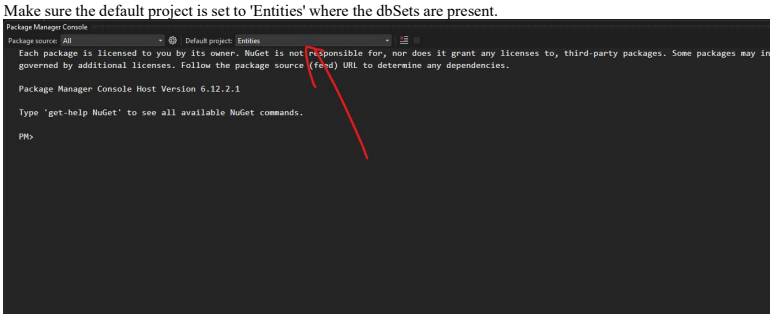
Migrations, also called code-first migrations, are commands used to manage database schema changes in Entity Framework.
They allow you to:

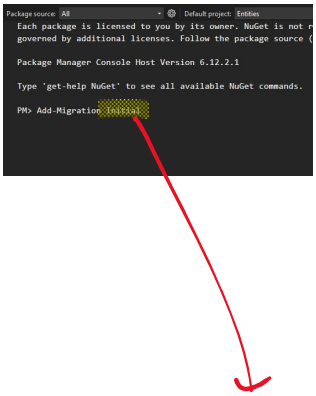
- Create tables initially.
- Modify existing table structures, such as adding a new column or changing a data type.

Each time you need to make structural changes to the database, you run a migration.
Fortunately, you don't need to write code manually for migrations. Instead, you run a simple command in the Package Manager Console, which generates the necessary C# migration code.
To apply the migration, you execute another command.
Let me demonstrate this practically.



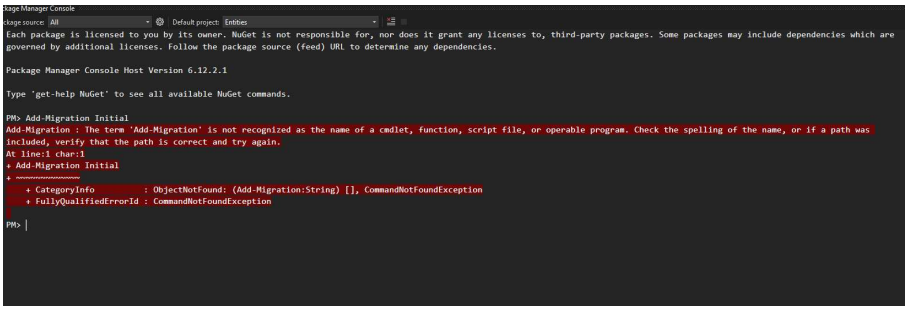
Make sure the default project is set to 'Entities' where the dbSets are present.





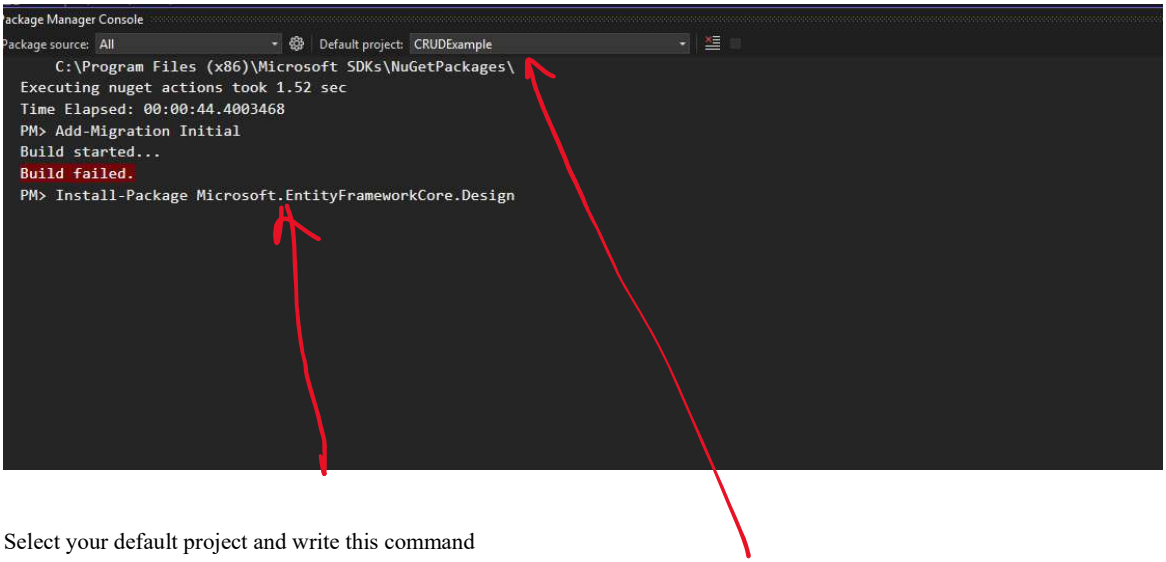
You can set any name other than 'Initial'

But we are getting error here, because in order to work with migration, we have to install another package.



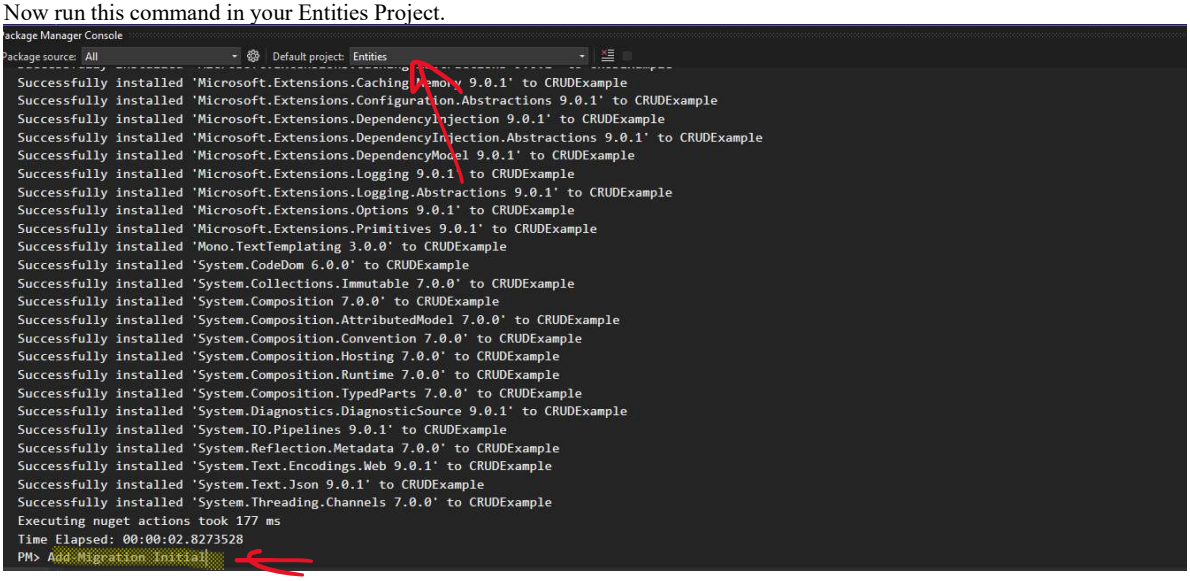
Install-Package Microsoft.EntityFrameworkCore.Tools

```
PM> Install-Package Microsoft.EntityFrameworkCore.Tools
Restoring packages for D:\ASSIGNMENT\Practice\DOT NET CORE\-Asp.Net-Core-9-.NET-9-True-Ultimate-Guide\Section 19 - EntityFrameworkCore [MVC and Web API]\Entities\Entiti
  GET https://pkgs.dev.azure.com/CSafeCloud/d1bc3433-4b39-472c-a998-d6b66227b272/_packaging/76aa53e0-7c36-4fed-984a-ba506cad1c0a/nuget/v3/flat2/microsoft.entityframewor
index.json
  GET https://api.nuget.org/v3-flatcontainer/microsoft.entityframeworkcore.tools/index.json
  OK https://api.nuget.org/v3-flatcontainer/microsoft.entityframeworkcore.tools/index.json 264ms
  GET https://api.nuget.org/v3-flatcontainer/microsoft.entityframeworkcore.tools/9.0.1/microsoft.entityframeworkcore.tools.9.0.1.nupkg
  OK https://api.nuget.org/v3-flatcontainer/microsoft.entityframeworkcore.tools/9.0.1/microsoft.entityframeworkcore.tools.9.0.1.nupkg 110ms
  GET https://api.nuget.org/v3-flatcontainer/microsoft.entityframeworkcore.design/index.json
  GET https://pkgs.dev.azure.com/CSafeCloud/d1bc3433-4b39-472c-a998-d6b66227b272/_packaging/76aa53e0-7c36-4fed-984a-ba506cad1c0a/nuget/v3/flat2/microsoft.entityframewor
index.json
  OK https://api.nuget.org/v3-flatcontainer/microsoft.entityframeworkcore.design/index.json 267ms
  GET https://api.nuget.org/v3-flatcontainer/microsoft.entityframeworkcore.design/9.0.1/microsoft.entityframeworkcore.design.9.0.1.nupkg
  OK https://api.nuget.org/v3-flatcontainer/microsoft.entityframeworkcore.design/9.0.1/microsoft.entityframeworkcore.design.9.0.1.nupkg 60ms
  NotFound https://pkgs.dev.azure.com/CSafeCloud/d1bc3433-4b39-472c-a998-d6b66227b272/_packaging/76aa53e0-7c36-4fed-984a-ba506cad1c0a/nuget/v3/flat2/microsoft.entityfra
index.json 2164ms
  GET https://api.nuget.org/v3-flatcontainer/microsoft.entityframeworkcore.relational/index.json
  GET https://pkgs.dev.azure.com/CSafeCloud/d1bc3433-4b39-472c-a998-d6b66227b272/_packaging/76aa53e0-7c36-4fed-984a-ba506cad1c0a/nuget/v3/flat2/microsoft.entityframewor
index.json
  GET https://api.nuget.org/v3-flatcontainer/microsoft.extensions.dependencymodel/index.json
```



Select your default project and write this command

Now run this command in your Entities Project.



We are getting another error. It says no database provider has been configured for this dbContext. You can configure the database provider either sql server or mysql or something else by calling the add dbContext while adding the dbContext class to the services collection.

```
Package Manager Console
Package source: All    Default project: Entities
  Successfully installed 'System.CodeDom 6.0.0' to CRUDExample
  Successfully installed 'System.Collections.Immutable 7.0.0' to CRUDExample
  Successfully installed 'System.Composition 7.0.0' to CRUDExample
  Successfully installed 'System.Composition.AttributedModel 7.0.0' to CRUDExample
  Successfully installed 'System.Composition.Convention 7.0.0' to CRUDExample
  Successfully installed 'System.Composition.Hosting 7.0.0' to CRUDExample
  Successfully installed 'System.Composition.Runtime 7.0.0' to CRUDExample
  Successfully installed 'System.Composition.TypedParts 7.0.0' to CRUDExample
  Successfully installed 'System.Diagnostics.DiagnosticSource 9.0.1' to CRUDExample
  Successfully installed 'System.IO.Pipelines 9.0.1' to CRUDExample
  Successfully installed 'System.Reflection.Metadata 7.0.0' to CRUDExample
  Successfully installed 'System.Text.Encodings.Web 9.0.1' to CRUDExample
  Successfully installed 'System.Text.Json 9.0.1' to CRUDExample
  Successfully installed 'System.Threading.Channels 7.0.0' to CRUDExample
  Executing nuget actions took 177 ms
  Time Elapsed: 00:00:02.8273528
  PM> Add-Migration Initial
  Build started...
  Build succeeded.
  An error occurred while accessing the Microsoft.Extensions.Hosting services. Continuing without the application service provider. Error: 'AddDbContext' was called with configuration, but
  the context type 'PersonsDbContext' only declares a parameterless constructor. This means that the configuration passed to 'AddDbContext' will never be used. If configuration is passed to
  'AddDbContext', then 'PersonsDbContext' should declare a constructor that accepts a DbContextOptions<PersonsDbContext> and must pass it to the base constructor for DbContext.
  Unable to create a 'DbContext' of type 'PersonsDbContext'. The exception 'No database provider has been configured for this DbContext. A provider can be configured by overriding the
  'DbContext.OnConfiguring' method or by using 'AddDbContext' on the application service provider. If 'AddDbContext' is used, then also ensure that your DbContext type accepts a
  DbContextOptions<TContext> object in its constructor and passes it to the base constructor for DbContext.' was thrown while attempting to create an instance. For the different patterns
  supported at design time, see https://go.microsoft.com/fwlink/?linkid=851728
  PM>
```

But we have already set the sql server (database provider) and connection string. These options will be supplied to the constructor of the db context class as options parameter.

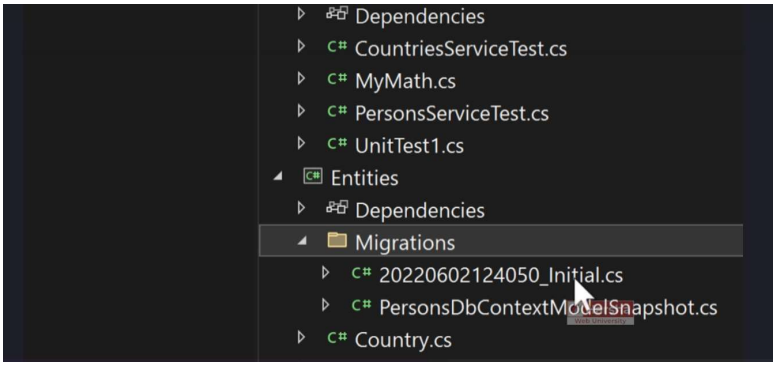
```
12
13  builder.Services.AddDbContext<PersonsDbContext>(options =>
14  {
15      //means, hey asp dot net core, we are trying to use asp dot net core for database connection
16      options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection"));
17  });
```

```
2   using System.Collections.Generic;
3   using Microsoft.EntityFrameworkCore;
4
5   namespace Entities
6   {
        2 references
7       public class PersonsDbContext : DbContext
8       {
            0 references
9           public PersonsDbContext(DbContextOptions options) : base
            (options)
10          {
11
12          }
```

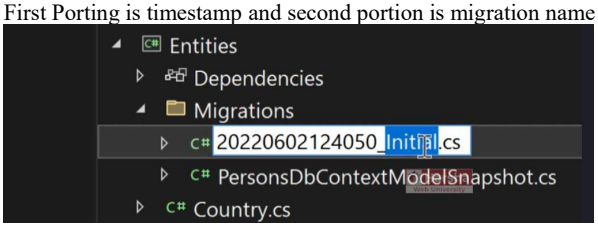
Now build your solution and run this same command.

```
Package Manager Console
Package source: All    Default project: Entities
  An error occurred while accessing the Microsoft.Extensions.Hosting services. Continuing without the application service provider.
  'PersonsDbContext' only declares a parameterless constructor. This means that the configuration passed to 'AddDbContext' will neve
  should declare a constructor that accepts a DbContextOptions<PersonsDbContext> and must pass it to the base constructor for DbCont
  Unable to create a 'DbContext' of type 'PersonsDbContext'. The exception 'No database provider has been configured for this DbCont
  method or by using 'AddDbContext' on the application service provider. If 'AddDbContext' is used, then also ensure that your DbCon
  passes it to the base constructor for DbContext.' was thrown while attempting to create an instance. For the different patterns su
  PM> Add-Migration Initial
```

Now build is successful. Under Migrations folder, we have got 2 files.



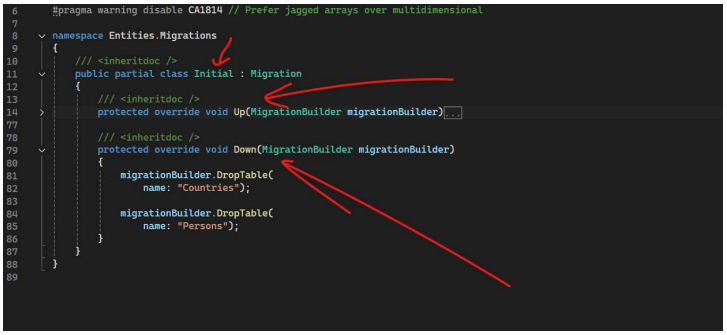
First Porting is timestamp and second portion is migration name that we have given.



Class name is name as your migration name.

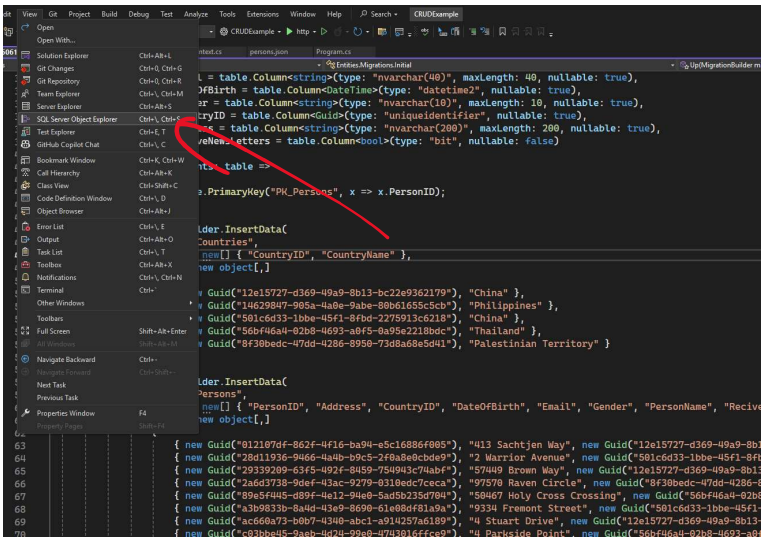
Up() method gets executed while creating the database or while making the changes into the existing.

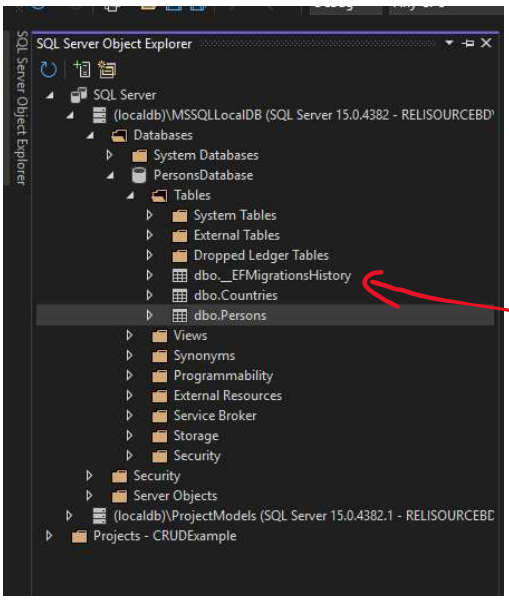
But in case if you rollback the migrations, the Down() method gets execute



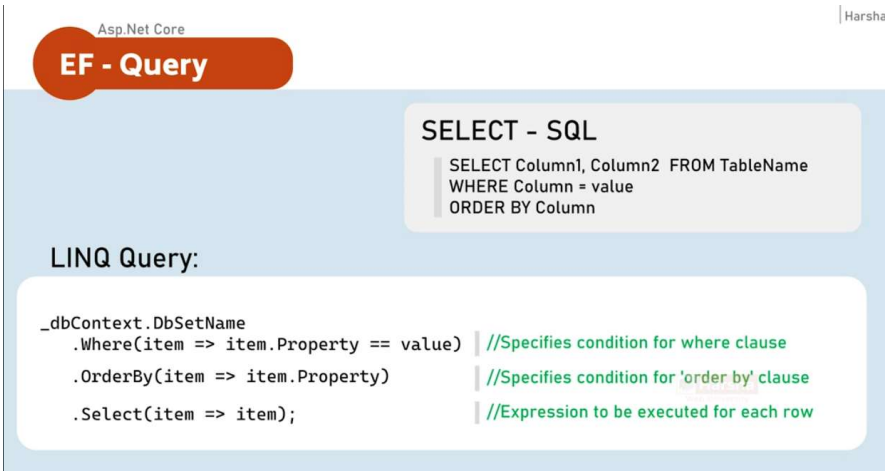
Now run this command,

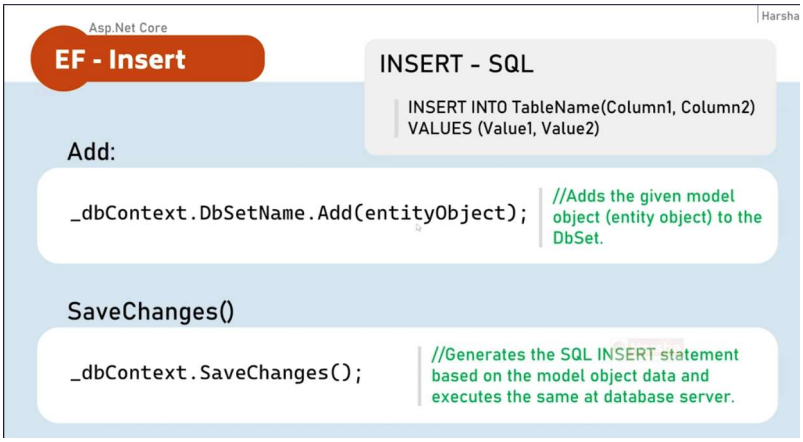
To see the table,

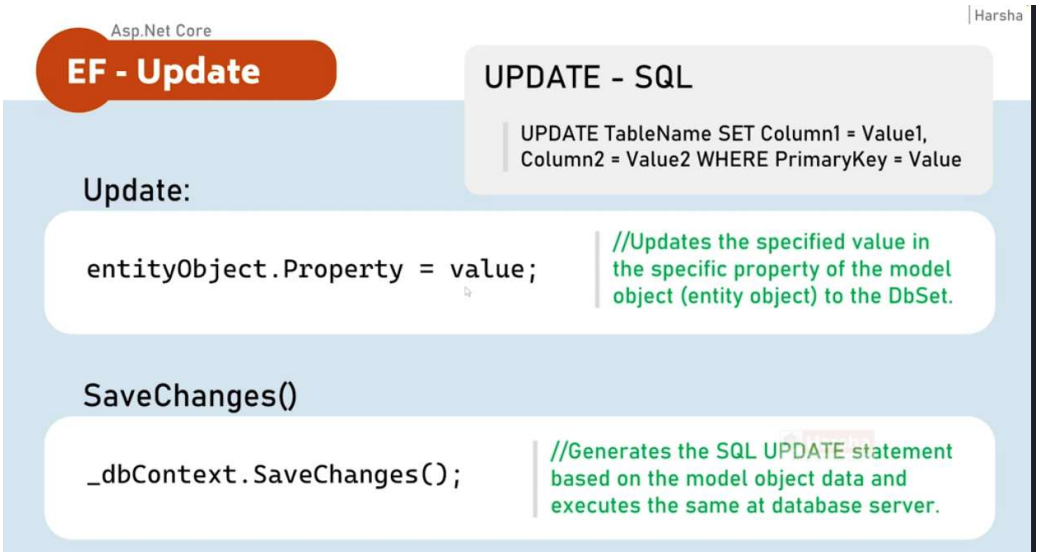


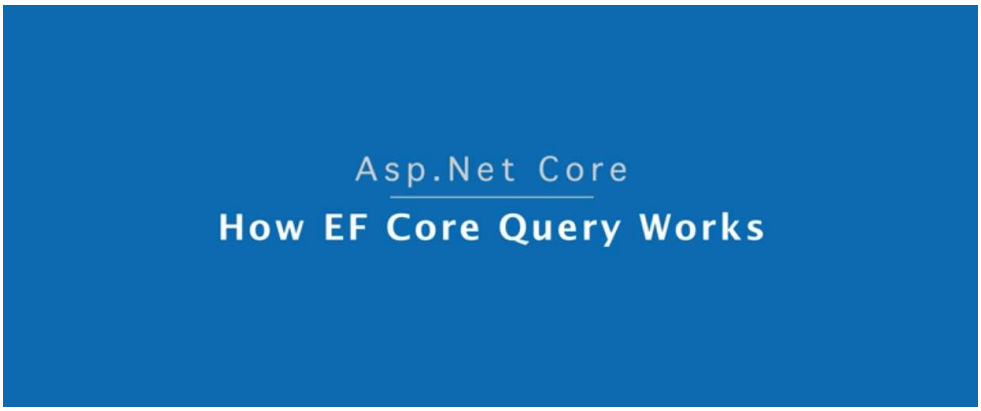


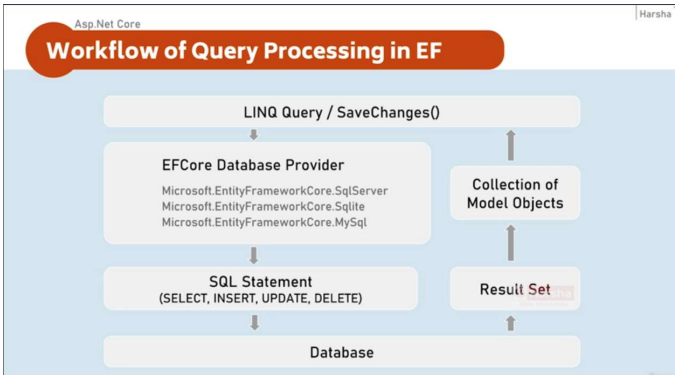
It stores what migrations are store so far.


Harsha
Asp.Net Core
EF - Query
SELECT - SQL
SELECT Column1, Column2 FROM TableName
WHERE Column = value
ORDER BY Column
LINQ Query:
_dbContext.DbSetName
.Where(item => item.Property == value) //Specifies condition for where clause
.OrderBy(item => item.Property) //Specifies condition for 'order by' clause
.Select(item => item); //Expression to be executed for each row


Harsha
Asp.Net Core
EF - Insert
INSERT - SQL
INSERT INTO TableName(Column1, Column2)
VALUES (Value1, Value2)
Add:
_dbContext.DbSetName.Add(entityObject); //Adds the given model object (entity object) to the DbSet.
SaveChanges()
_dbContext.SaveChanges(); //Generates the SQL INSERT statement based on the model object data and executes the same at database server.


Harsha
Asp.Net Core
EF - Update
UPDATE - SQL
UPDATE TableName SET Column1 = Value1,
Column2 = Value2 WHERE PrimaryKey = Value
Update:
entityObject.Property = value; //Updates the specified value in the specific property of the model object (entity object) to the DbSet.
SaveChanges()
_dbContext.SaveChanges(); //Generates the SQL UPDATE statement based on the model object data and executes the same at database server.





If you want to perform multiple or complex database operations, meaning you would like to execute multiple SQL statements at a time with a single database call, then creating a stored procedure in the database is the best choice. With stored procedures, you will **group multiple SQL statements** as one stored unit, and you can call the same from your application source code with a **single database call**.

Asp.Net Core

EF - Calling Stored Procedures

Harsha

Stored Procedure for CUD (INSERT | UPDATE | DELETE):

```
int DbContext.Database.ExecuteSqlRaw(
   string sql,                    //Eg: "EXECUTE [dbo].[StoredProcName]  @Param1 @Parm2"
   params object[] parameters)    //A list of objects of SqlParameter type
```

Stored Procedure for Retrieving (Select):

```
IQueryable<Model> DbSetName.FromSqlRaw(
   string sql,                    //Eg: "EXECUTE [dbo].[StoredProcName]  @Param1 @Parm2"
   params object[] parameters)    //A list of objects of SqlParameter type
```

```
licenses. Follow the package source (feed) URL to determine any dependencies.

Package Manager Console Host Version 6.12.2.1

Type 'get-help NuGet' to see all available NuGet commands.

PM> Add-Migration GetPersons_StoredProcedure
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM>
```

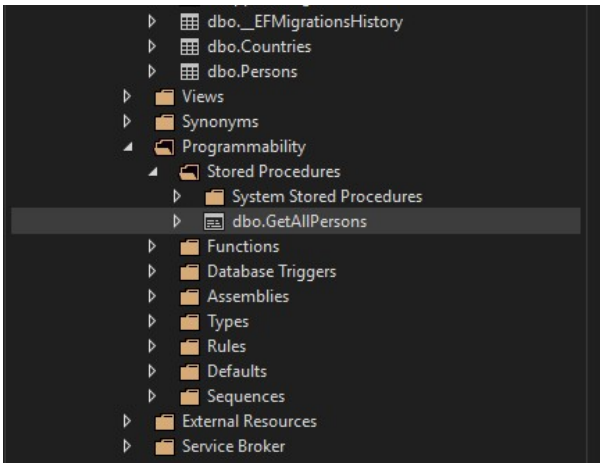
```
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace Entities.Migrations
{
    /// <inheritdoc />
    public partial class GetPersonsStoredProcedure : Migration
    {
        /// <inheritdoc />
        protected override void Up(MigrationBuilder migrationBuilder)
        {

        }

        /// <inheritdoc />
        protected override void Down(MigrationBuilder migrationBuilder)
        {

        }
    }
}
```

```
/// <inheritdoc />
protected override void Up(MigrationBuilder migrationBuilder)
{
    string sp_GetAllPersons = @"
        CREATE PROCEDURE [dbo].[GetAllPersons]
        AS BEGIN
            SELECT PersonID, PersonName, Email, DateOfBirth, Gender, CountryID, Address, ReceiveNewsLet
            FROM [dbo].[Persons]
        END
    ";

    migrationBuilder.Sql(sp_GetAllPersons);
}

/// <inheritdoc />
protected override void Down(MigrationBuilder migrationBuilder)
{
    string sp_GetAllPersons = @"
        DROP PROCEDURE [dbo].[GetAllPersons]";

    migrationBuilder.Sql(sp_GetAllPersons);
```

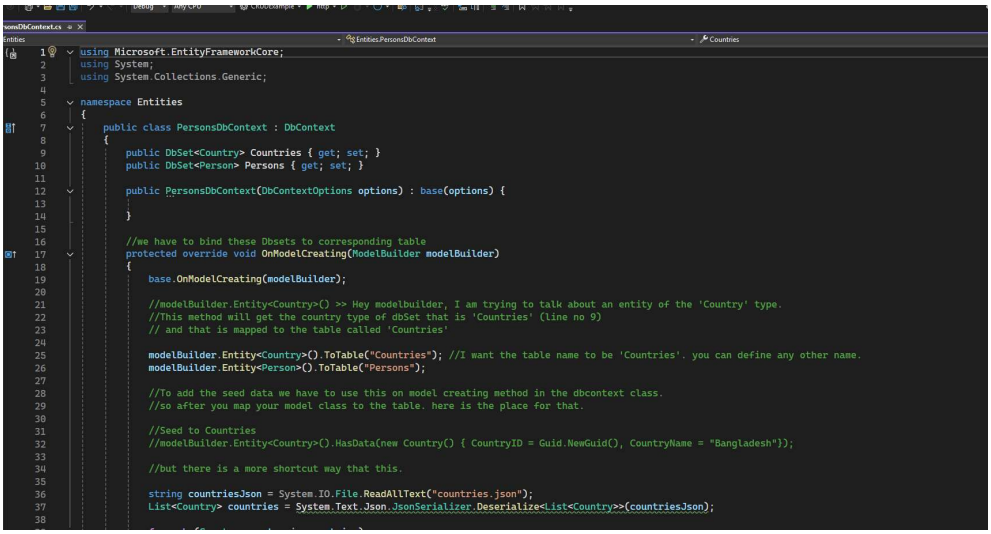
Now run this command.

```
To undo this action, use Remove-Migration.
PM> Update-Database
Build started...
Build succeeded.
Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (10ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT 1
```

Here you can see you stored procedure.

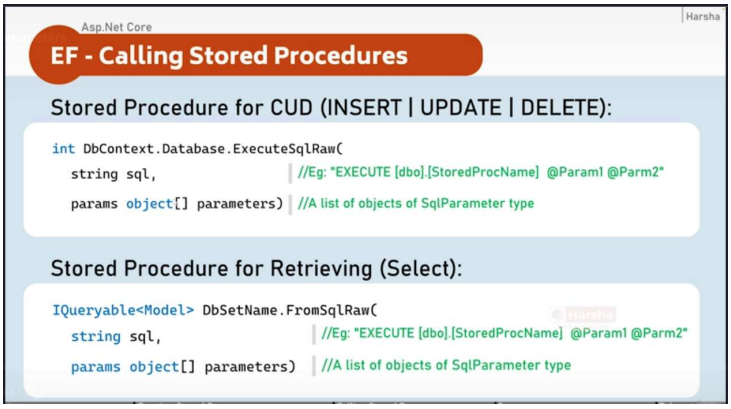


Nome come back here and create your stored procedure method.

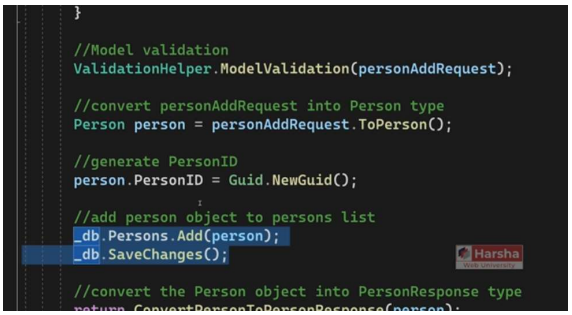


```cs
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;

namespace Entities
{
    public class PersonsDbContext : DbContext
    {
        public DbSet<Country> Countries { get; set; }
        public DbSet<Person> Persons { get; set; }

        public PersonsDbContext(DbContextOptions options) : base(options) {

        }

        //we have to bind these Dbsets to corresponding table
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);

            //modelBuilder.Entity<Country>() >> Hey modelbuilder, I am trying to talk about an entity of the 'Country' type.
            //This method will get the country type of dbSet that is 'Countries' (line no 9)
            // and that is mapped to the table called 'Countries'

            modelBuilder.Entity<Country>().ToTable("Countries"); //I want the table name to be 'Countries'. you can define any other name.
            modelBuilder.Entity<Person>().ToTable("Persons");

            //To add the seed data we have to use this on model creating method in the dbcontext class.
            //so after you map your model class to the table. here is the place for that.

            //Seed to Countries
            //modelBuilder.Entity<Country>().HasData(new Country() { CountryID = Guid.NewGuid(), CountryName = "Bangladesh"});

            //but there is a more shortcut way that this.

            string countriesJson = System.IO.File.ReadAllText("countries.json");
            List<Country> countries = System.Text.Json.JsonSerializer.Deserialize<List<Country>>(countriesJson);
```

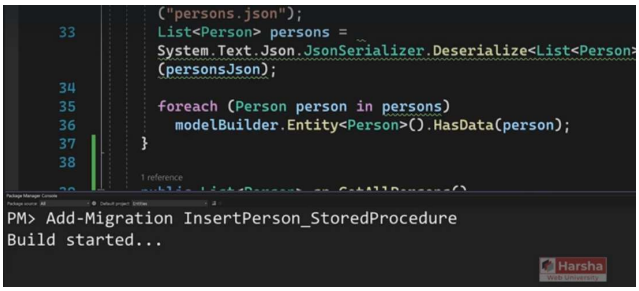
How can you pass parameter to the stored procedure?

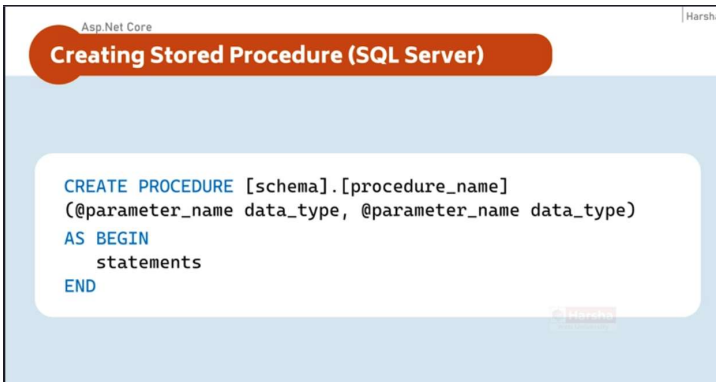


Instead of adding like this, let's use stored procedure to insert data through stored procedure.

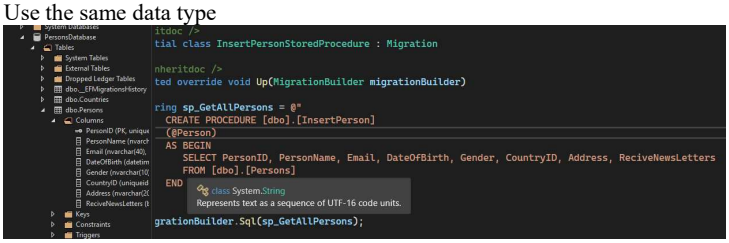


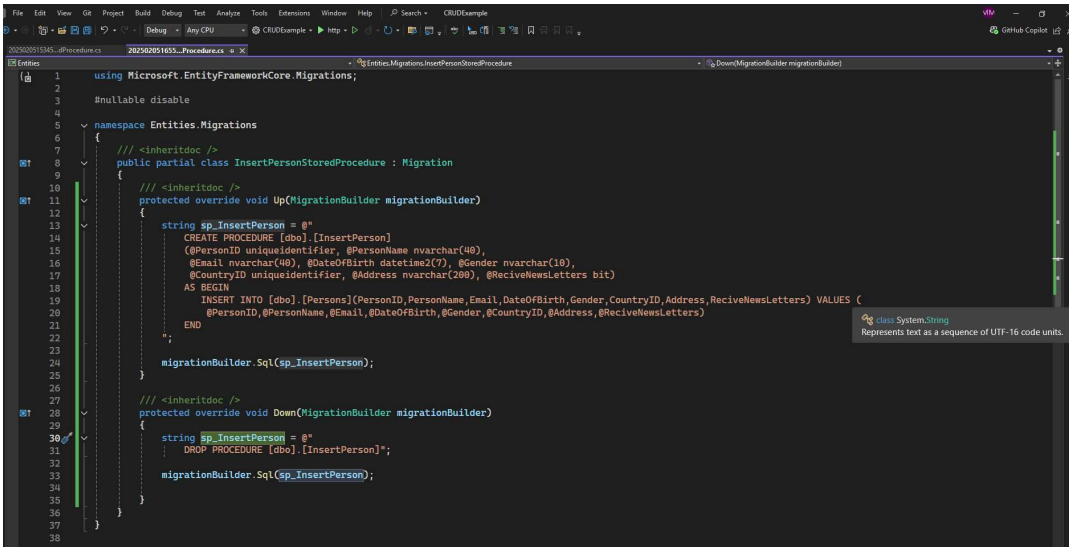
Create a migration file





Use the same data type



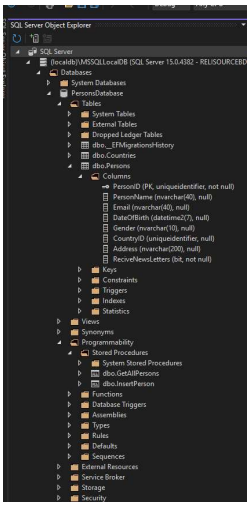


```
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace Entities.Migrations
{
    /// <inheritdoc />
    public partial class InsertPersonStoredProcedure : Migration
    {
        /// <inheritdoc />
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            string sp_InsertPerson = @"
                CREATE PROCEDURE [dbo].[InsertPerson]
                (@PersonID uniqueidentifier, @PersonName nvarchar(40),
                 @Email nvarchar(40), @DateOfBirth datetime2(7), @Gender nvarchar(10),
                 @CountryID uniqueidentifier, @Address nvarchar(200), @ReciveNewsLetters bit)
                AS BEGIN
                    INSERT INTO [dbo].[Persons](PersonID,PersonName,Email,DateOfBirth,Gender,CountryID,Address,ReciveNewsLetters) VALUES (
                     @PersonID,@PersonName,@Email,@DateOfBirth,@Gender,@CountryID,@Address,@ReciveNewsLetters)
                END
            ";

            migrationBuilder.Sql(sp_InsertPerson);
        }

        /// <inheritdoc />
        protected override void Down(MigrationBuilder migrationBuilder)
        {
            string sp_InsertPerson = @"
                DROP PROCEDURE [dbo].[InsertPerson]";

            migrationBuilder.Sql(sp_InsertPerson);

        }
    }
}
```

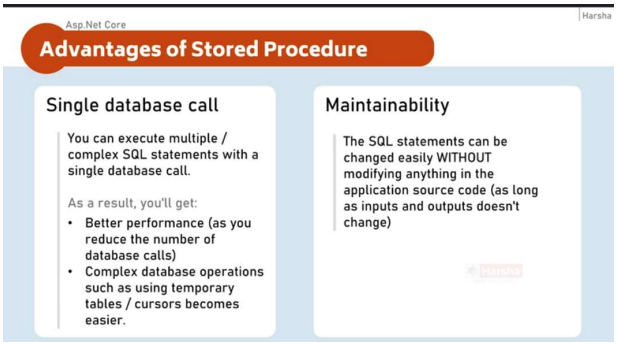
class System.String
Represents text as a sequence of UTF-16 code units.

Now go to package manager console and run this command.

Update-Database



At your practice time, you can try update and delete procedure.



Asp.Net Core

Changes in Table Structure

Once the table has been created, how do you make changes to the existing database table?
How do you change the table structure?

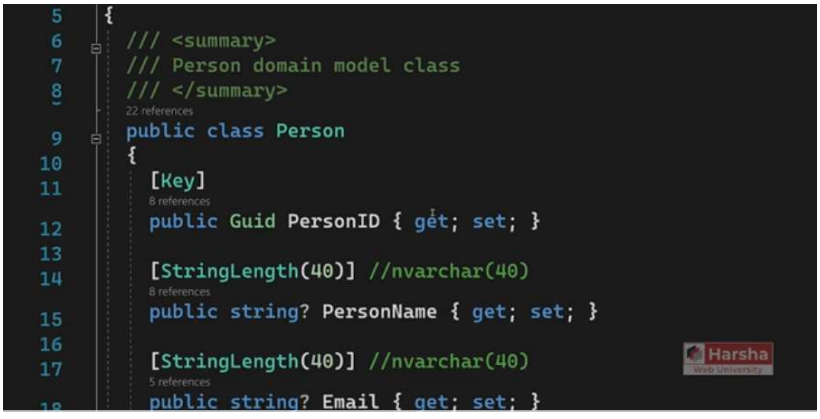
For example, you may want to rename a column, change its data type, or add some constraints.
This is exactly what we are going to perform in the upcoming lectures.

In this lecture, let us try to learn how to add or remove columns.
For example, in the **Person** table, these are all the existing columns.

Now, I would like to add an additional column.
In the **Person** model class, we are going to add a new column, for example, **TIN** (Tax Identification Number).

Assume it is an alphanumerical value, so we are using the **string** data type, and of course, it is nullable, meaning the values can be **null** in the database table.
Since we have added this new column directly in the model class, these changes should be reflected in the actual database table.

But it's not magic—it doesn't happen automatically.
See, in the **Persons** table, if we refresh, there are no changes.
So, we have to apply **migrations** in order to create that column in the table.
But here, instead of writing the actual SQL script to add the column, we are letting the **migrations system** generate it for us.



```
23      public string? Gender { get; set; }
24
25      //uniqueidentifier
        6 references
26      public Guid? CountryID { get; set; }
27
28      [StringLength(200)] //nvarchar(200)
        5 references
29      public string? Address { get; set; }
30
31      //bit
        5 references
32      public bool ReceiveNewsLetters { get; set; }
33
        0 references
34      public string? TIN { get; set; }
35    }
36  }
37
```

Newly Added

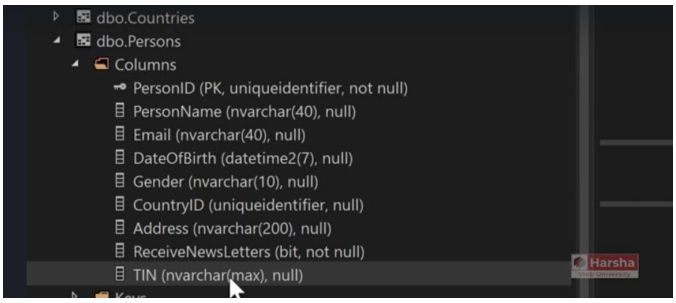
- dbo.__EFMigrationsHistory
- dbo.Countries
- dbo.Persons
  - Columns
    - PersonID (PK, uniqueidentifier, not null)
    - PersonName (nvarchar(40), null)
    - Email (nvarchar(40), null)
    - DateOfBirth (datetime2(7), null)
    - Gender (nvarchar(10), null)
    - CountryID (uniqueidentifier, null)
    - Address (nvarchar(200), null)
    - ReceiveNewsLetters (bit, not null)
  - Keys

Now go to your package manager console.

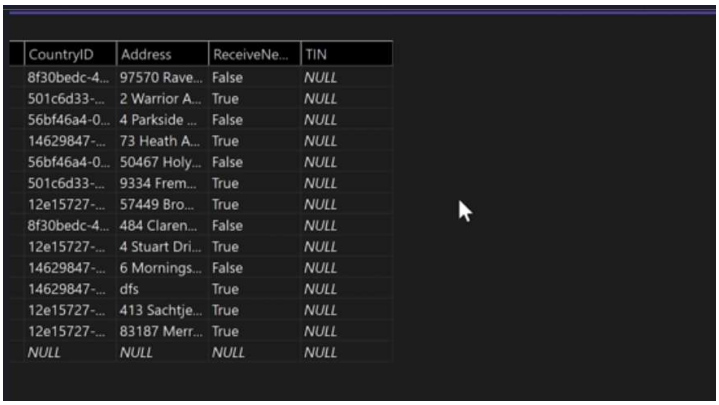
```
      INSERT INTO [__EFMigrationsHi
[ProductVersion])
      VALUES (N'20220611123233_Inse
N'6.0.5');
Done.
PM> Add-Migration TINColumn
```

```
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace Entities.Migrations
{
    /// <inheritdoc />
    public partial class TINColumn : Migration
    {
        /// <inheritdoc />
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.AddColumn<string>(
                name: "TIN",
                table: "Persons",
                type: "nvarchar(max)",
                nullable: true);

            migrationBuilder.UpdateData(
                table: "Persons",
                keyColumn: "PersonID",
                keyValue: new Guid("012107df-862f-4f16-ba94-e5c16886f005"),
                column: "TIN",
                value: null);

            migrationBuilder.UpdateData(
                table: "Persons",
                keyColumn: "PersonID",
                keyValue: new Guid("28d11936-9466-4a4b-b9c5-2f0a8e0cbde9"),
                column: "TIN",
                value: null);

            migrationBuilder.UpdateData(
                table: "Persons",
                keyColumn: "PersonID",
                keyValue: new Guid("29339209-63f5-492f-8459-754943c74abf"),
                column: "TIN",
```

```
To undo this action, use Remove-Migration.
PM> Update-Database
Build started...
Build succeeded.
```

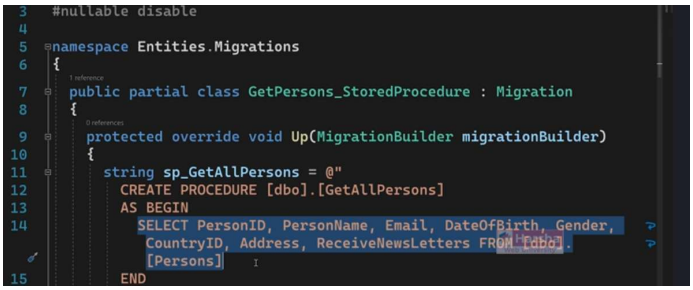


Now, your assignment is to create a new migration to set default value for TIN

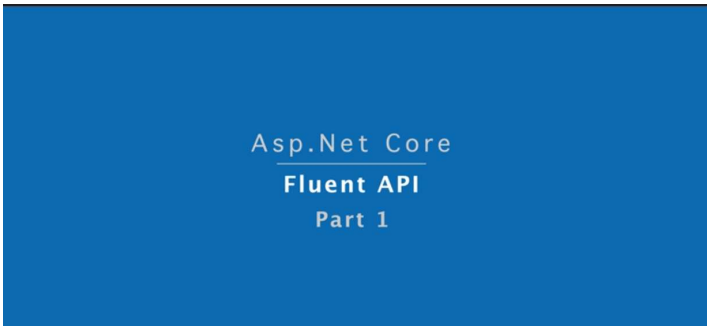


CountryID	Address	ReceiveNe...	TIN
8f30bedc-4...	97570 Rave...	False	NULL
501c6d33-...	2 Warrior A...	True	NULL
56bf46a4-0...	4 Parkside ...	False	NULL
14629847-...	73 Heath A...	True	NULL
56bf46a4-0...	50467 Holy...	False	NULL
501c6d33-...	9334 Frem...	True	NULL
12e15727-...	57449 Bro...	True	NULL
8f30bedc-4...	484 Claren...	False	NULL
12e15727-...	4 Stuart Dri...	True	NULL
14629847-...	6 Mornings...	False	NULL
14629847-...	dfs	True	NULL
12e15727-...	413 Sachtje...	True	NULL
12e15727-...	83187 Merr...	True	NULL
NULL	NULL	NULL	NULL

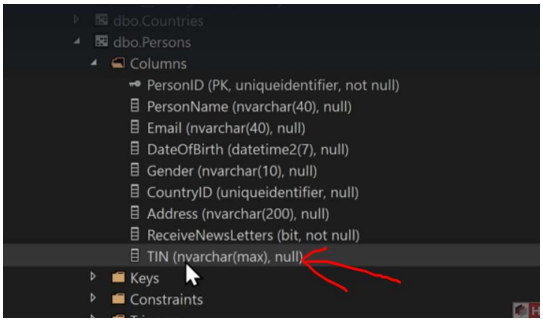
Now run our application, we have got an error. Because in previous SP, we have not set it.



```
#nullable disable

namespace Entities.Migrations
{
    public partial class GetPersons_StoredProcedure : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            string sp_GetAllPersons = @"
              CREATE PROCEDURE [dbo].[GetAllPersons]
              AS BEGIN
                SELECT PersonID, PersonName, Email, DateOfBirth, Gender,
                CountryID, Address, ReceiveNewsLetters FROM [dbo].
                [Persons]
              END
```



```
/// Person domain model class
/// </summary>
public class Person
{
    [Key]
    public Guid PersonID { get; set; }

    //default data type in database is : nvarchar(max)

    [StringLength(40)] //nvarchar(40)
    public string? PersonName { get; set; }

    [StringLength(40)]
    public string? Email { get; set; }

    public DateTime? DateOfBirth { get; set; }

    [StringLength(10)]
    public string? Gender { get; set; }

    //data type: uniqueidentifier
    //It's a foreign key. that means it refer to the Country Table.
    public Guid? CountryID { get; set; }

    [StringLength(200)]
    public string? Address { get; set; }

    //data type: bit (no need to mention any other data type)
    public bool ReciveNewsLetters { get; set; }

    public string? TIN { get; set;}
}
}
```



Okay, in the last lecture, we added a new column called **TIN** in the **Persons** table.
First, we added a property called **TIN** in the **Person** class.

Then, we ran the **Add-Migration** and **Update-Database** commands in the **Package Manager Console**.
This generated the migration and added that column to the **Persons** table.

Now, if you check **SQL Server Object Explorer**, expand **Databases**, and navigate to the **Persons** table, you will see that it has the **TIN** column.

However, we have a big complaint about its **data type**.
It is **nvarchar(max)**, which means it supports the maximum number of characters allowed—up to **2 billion characters** in a single string.

Moreover, **nvarchar** is a Unicode data type, meaning it supports storing **alphabetical, numerical, and special characters** from any language supported by Unicode.

While this sounds good, it has a **negative impact on database performance** because it dramatically **increases the database size**.
Each character in **nvarchar** occupies **two bytes** instead of one byte (as in ASCII).

Therefore, you should **not** use the **nvarchar** data type unless it's absolutely necessary.
For example, in the case of a **person's name**, you might require **nvarchar** because names can contain **non-numerical characters**, such as those from different languages.

But in this case, the **TIN number** consists only of **alphabets and digits**, so it is **not advisable** to use **nvarchar**.
Now, the question is:

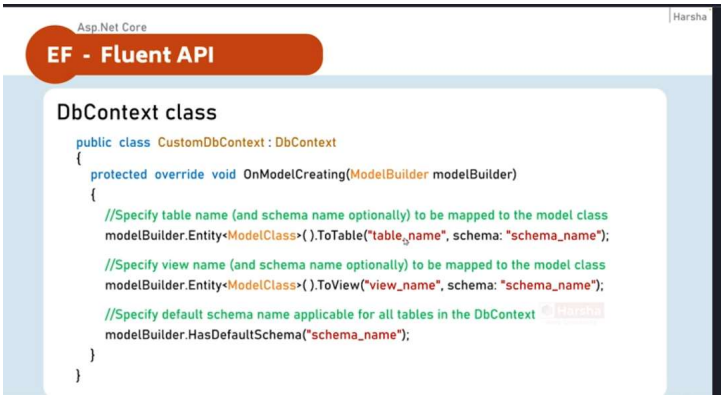
How can we control the SQL Server data type from Entity Framework?
How do we specify the **exact** data type for a specific column in the database?
This is where the **Fluent API** comes into the picture.

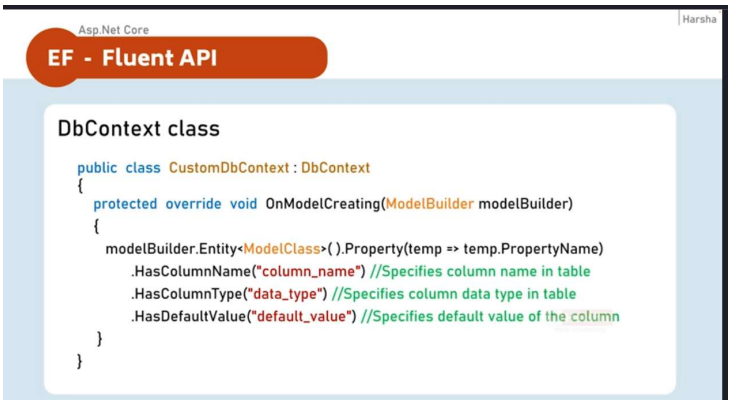
The **Fluent API** is a set of predefined methods in **Entity Framework** that allows you to specify more details about **column data types** and table structures.
This includes:

- **Table name**
- **Column names**
- **Data types**
- **Constraints**

You can use the **Fluent API** inside the **OnModelCreating** method of the **DbContext** class.

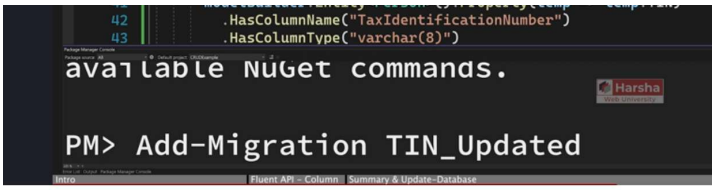
In fact, we are already using it with the **ToTable** method to specify the table name that is mapped to a specific **model class**.



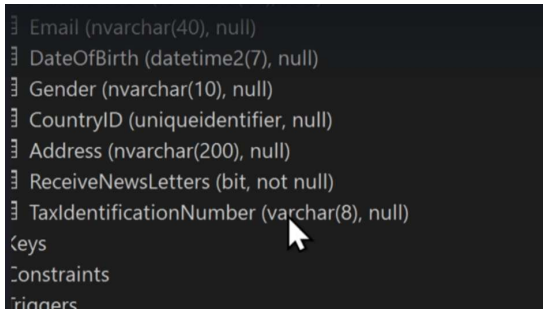


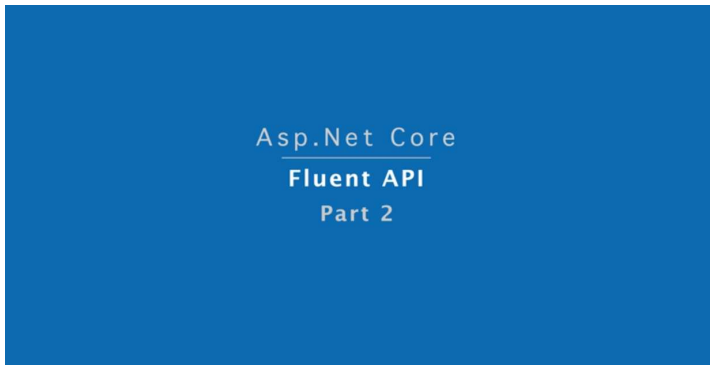
Go to your PersonsDbContext class.

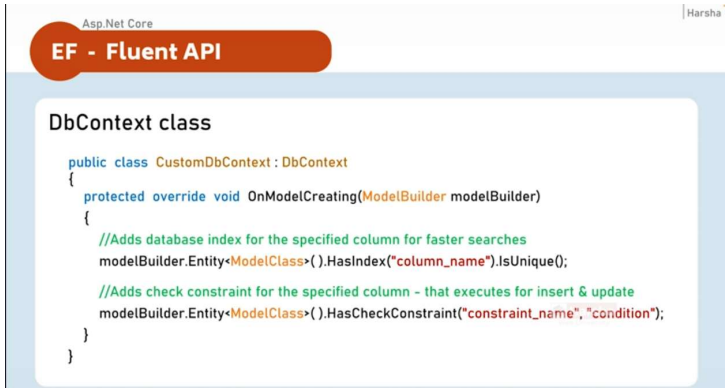
Let's add migration.



Now run this command.





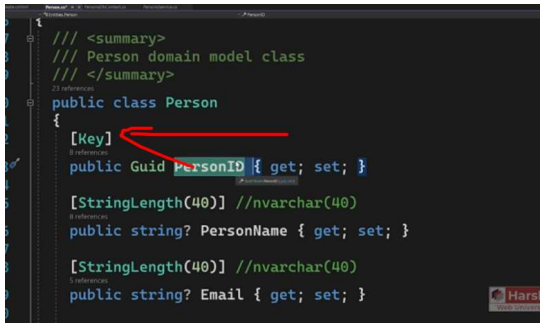


To add **constraints** to a database table using **Entity Framework Fluent API**, you can define them inside the **OnModelCreating** method of the **DbContext** class.

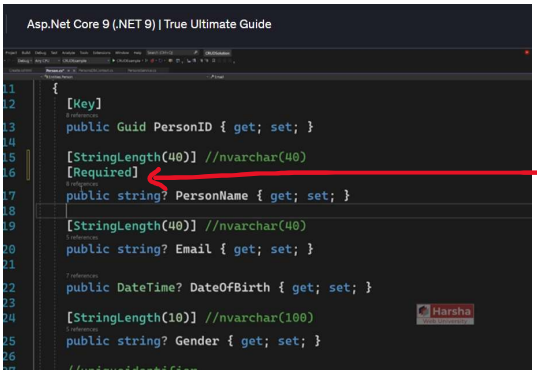
Common Constraints in Fluent API:

1. **Primary Key**
2. **Foreign Key**
3. **Unique Constraint**
4. **Required (NOT NULL)**
5. **Default Values**
6. **Column Length and Type**

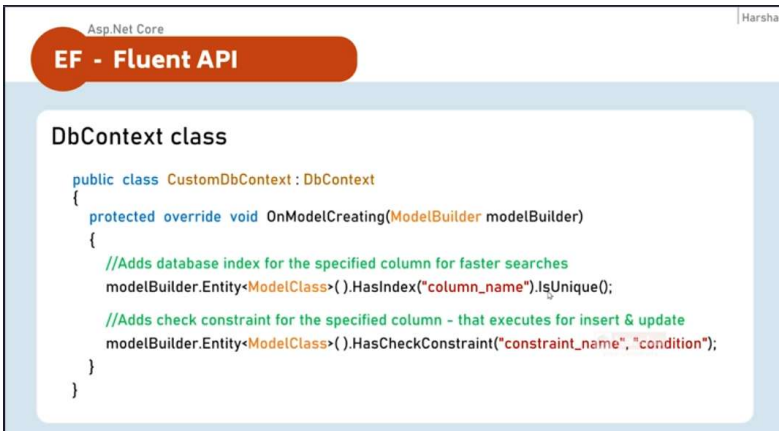
See, the first constraint is the primary key constraint, but we have already mentioned the same by using the key attribute for the person ID, so this is your primary key. That means it is not NULL + UNIQUE



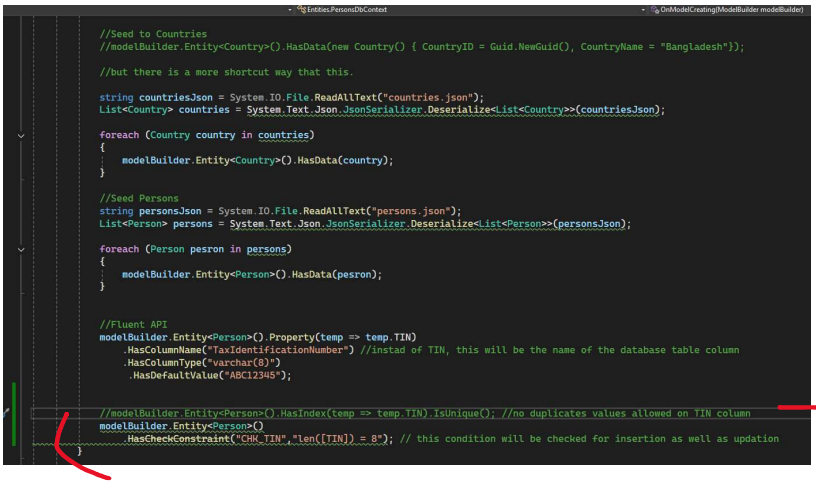
But to add the not null constraint, we will add the required attribute. For example, if you add a required attribute for a particular column, that means it has a not null constraint. Okay, we don't want to add it right now.



But how to add the unique constraint and check constraint we can do that through Fluent API.

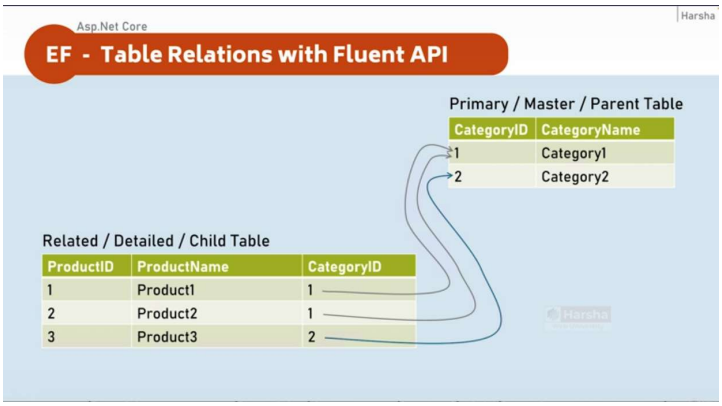


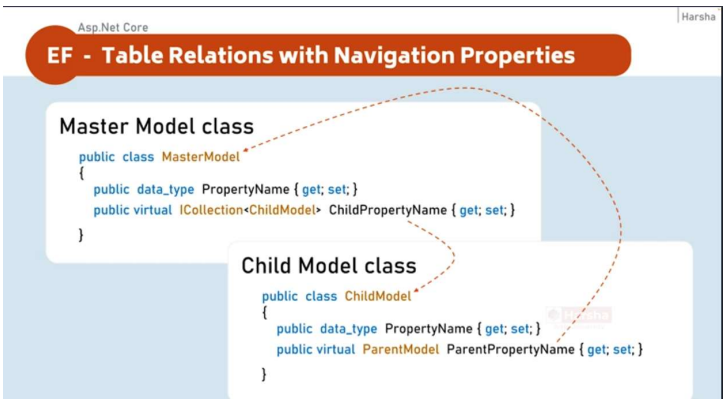
In the "PersonsDbContext"



Now run this command in the package manager console.

```
Type 'get-help NuGet' to see all available NuGet commands.

PM> Add-Migration TIN_Updated_CHK\
Build started...
Build succeeded.
Your target project 'CRUDExample' doesn't match your migrations assembly 'Entities'. Either change your targ
Change your migrations assembly by using DbContextOptionsBuilder. E.g. options.UseSqlServer(connection, b =>
the DbContext.
Change your target project to the migrations project by using the Package Manager Console's Default project
PM> Add-Migration TIN_Updated_CHK
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM> Update-Database
```





SQL Server referential integrity specifies how you separate the data between the primary table and detailed table.
In the primary table or master table, you will have the primary key, which contains all available values of the particular column.
For example, there are two tables like Categories and Products. The category_id column of the Categories table contains all available categories, so these values are referenced in the foreign key property of the Products table.

For example, for this product, category_id equals 1. That means, to get the category name of this, you have to search for the matching category_id value in the Categories table. There, you can find the name.

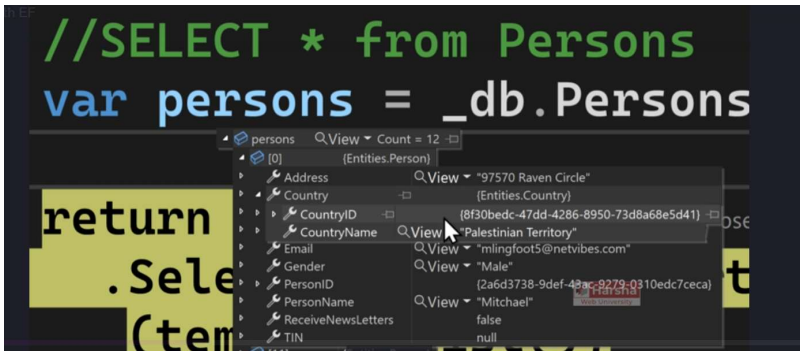
In that way, you have to match the foreign key value with the primary key value.
Okay, this is the general database-specific rule.
But how do you configure the primary key and foreign key?
In the same way, how can you access the related table data in Entity Framework?

That's what we are trying to learn in this lecture.
In Entity Framework, you can make use of navigation properties instead of joins.
For example, in the child model class, you can create a parent property that refers to the parent record.
For example, this is the Person model class.
So, it can have a property called Country, which refers to the corresponding country object—that is, an object of the Country model class.
In the same way, every country can refer to a set of persons. So, all the persons that come under a particular country can be referred to by this collection-type property, which is of ICollection type.

Here, ICollection is the parent interface for List.

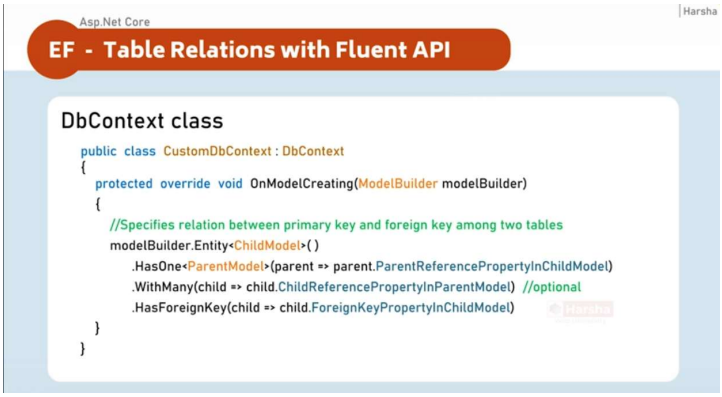
Optionally, you can make it virtual so that you can override it in the child classes.
Let me demonstrate the same.

```
31
32      public PersonResponse AddPerson(PersonAddRequest? personAddRequest)...
58
59      public List<PersonResponse> GetAllPersons()
60      {
61          //it is not possible to call your own method in the link to entites expression
62          //return _db.Persons.Select(person => ConvertPersonToPersonResponse(person)).ToList();
63
64          //return _db.Persons.ToList() //SELECT * from Persons
65          //         .Select(person => ConvertPersonToPersonResponse(person)).ToList();
66
67          var persons = _db.Persons.Include("Country").ToList();
68
69          return _db.sp_GetAllPersons()   ≤ 1,542ms elapsed
70                    .Select(person => ConvertPersonToPersonResponse(person)).ToList();
71      }
72
73      public PersonResponse? GetPersonByPersonId(Guid? personID)
```

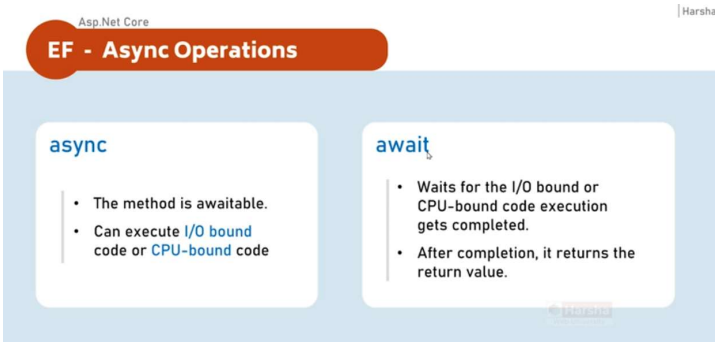


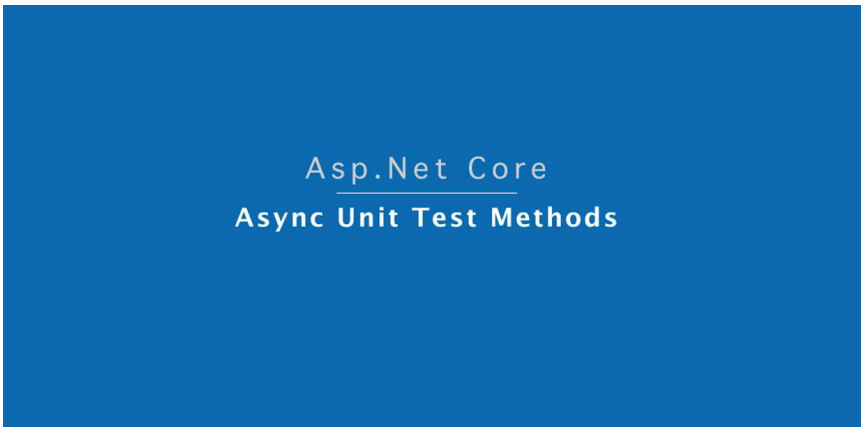

```
//default data type in database is : nvarchar(max)

[StringLength(40)] //nvarchar(40)
public string? PersonName { get; set; }

[StringLength(40)]
public string? Email { get; set; }

public DateTime? DateOfBirth { get; set; }

[StringLength(10)]
public string? Gender { get; set; }

//data type: uniqueidentifier
//It's a foreign key. that means it refer to the Country Table.
public Guid? CountryID { get; set; }

[StringLength(200)]
public string? Address { get; set; }

//data type: bit (no need to mention any other data type)
public bool ReciveNewsLetters { get; set; }

public string? TIN { get; set;}

[ForeignKey("CountryID")]
public virtual Country? Country { get; set; } //virtual so child class can override that. not mandatory.
}
}
```

Theirs another way to configure the table relations through fluent api.

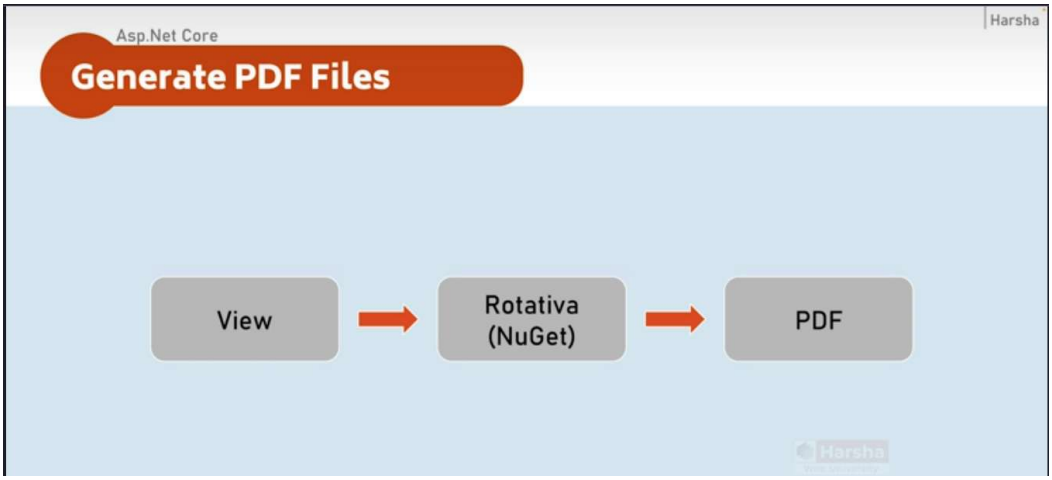


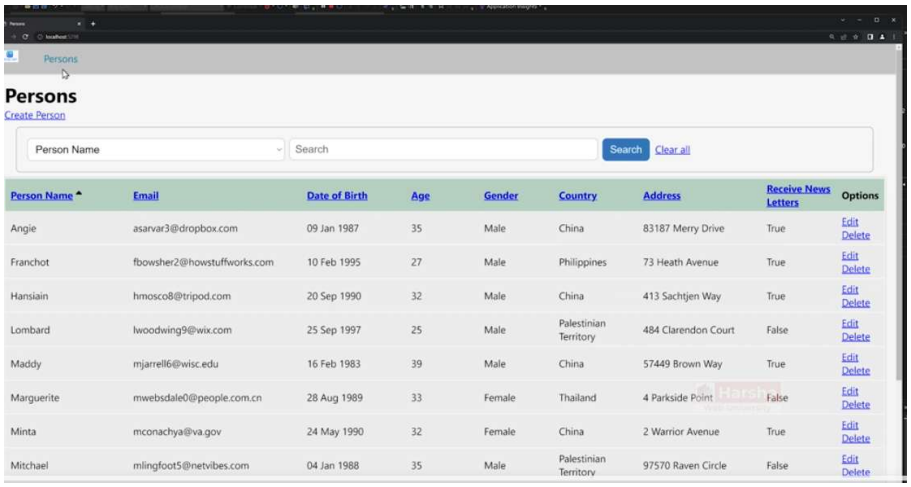
The asynchronous method execution can be paused by using the await keyword based on either I/O-bound code or CPU-bound code.
That means, as soon as the particular task has been completed, it can resume the execution of the asynchronous method.
Meanwhile, the same thread can execute some other method or handle another request.
In that way, you can ensure the maximum usage of CPU capacity.
First of all, you will create an asynchronous method by using a single keyword, and it can use one or more await keywords wherever you would like to wait for a response from a particular method.
For example, reading data from a file or database, or executing a large task.
Wherever you would like to wait for a particular method execution, you can use the await keyword.
Upon using the await keyword, the current method execution can be paused until we get the proper response from the task.
This is how asynchronous methods work in general in C#.
We have already used async and await keywords several times.
For example, we have used them in middleware.
Particularly in this lecture, we are going to use the async and await keywords in service methods as well as controller action methods.
In real-world projects, it is always advisable to maintain service methods and controller action methods as asynchronous methods.
Because most I/O operations, particularly database interactions, should be done with asynchronous operations.
This means you should always use async and await while interacting with files, databases, or any other external resources such as communicating with email servers, etc.

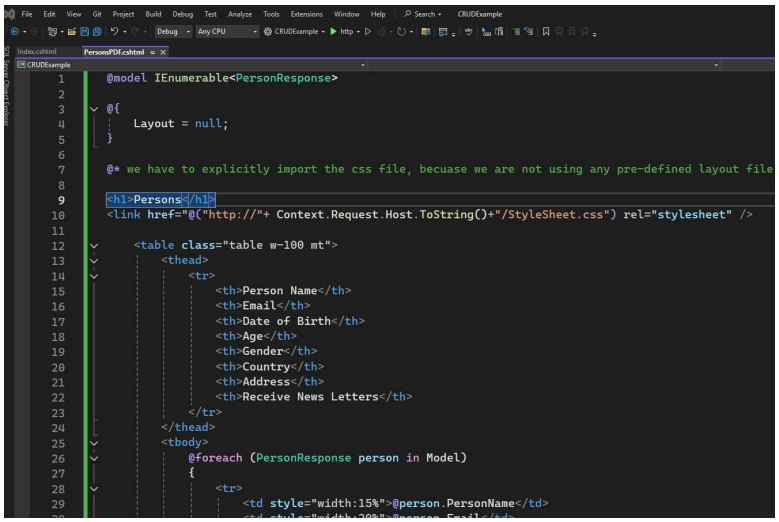




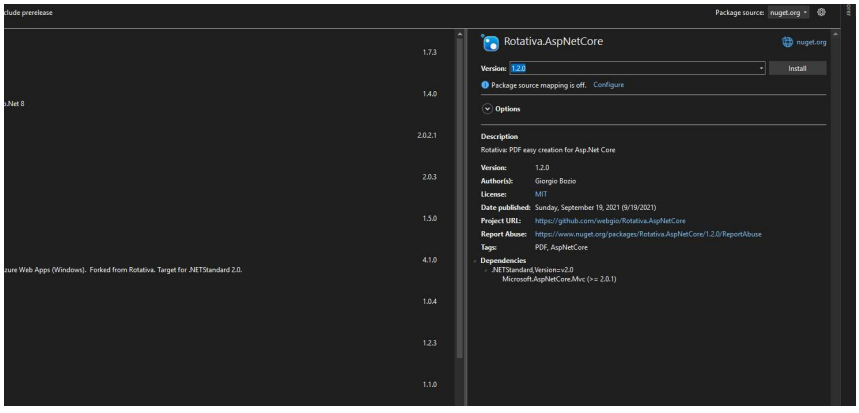
Suppose one of the client requirements is that you have to generate a PDF file programmatically.
For example, the client asks for a report or the data in the form of a PDF file.
For a real-world example, let's say there is a ticket booking application, and the ticket should be generated as a PDF for download.
Such requirements can be satisfied using a NuGet package called **Rotativa**.
It is one of the third-party packages but is pretty useful and commonly used for generating PDF files in ASP.NET Core.
All you need to do is create a view that contains the content.
That view should include all the content that needs to be in the PDF file.
Whatever information should be in the PDF file, you can write it in the view.
Then, you will use this NuGet package called **Rotativa** to convert the view into a PDF.
Let me demonstrate the same.



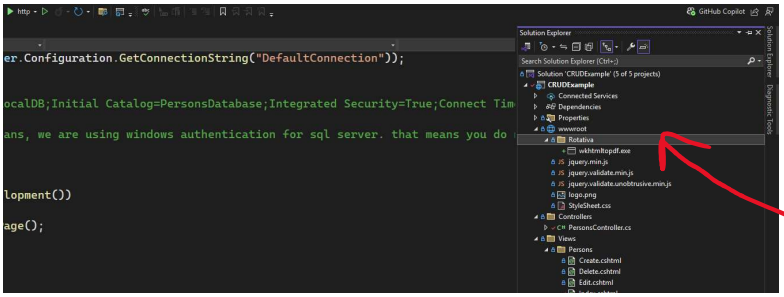




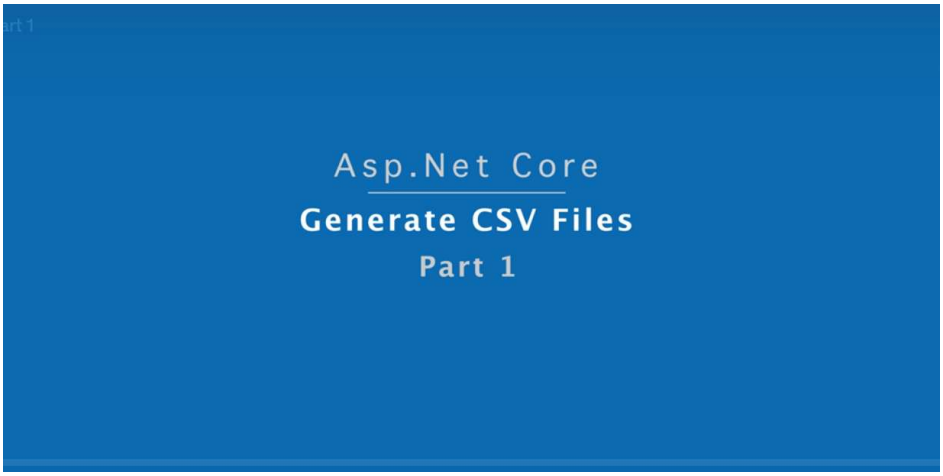
```
@model IEnumerable<PersonResponse>

@{
    Layout = null;
}

@* we have to explicitly import the css file, becuase we are not using any pre-defined layout file

<h1>Persons</h1>
<link href="@("http://"+ Context.Request.Host.ToString()+"/StyleSheet.css") rel="stylesheet" />

    <table class="table w-100 mt">
        <thead>
            <tr>
                <th>Person Name</th>
                <th>Email</th>
                <th>Date of Birth</th>
                <th>Age</th>
                <th>Gender</th>
                <th>Country</th>
                <th>Address</th>
                <th>Receive News Letters</th>
            </tr>
        </thead>
        <tbody>
            @foreach (PersonResponse person in Model)
            {
                <tr>
                    <td style="width:15%">@person.PersonName</td>
```

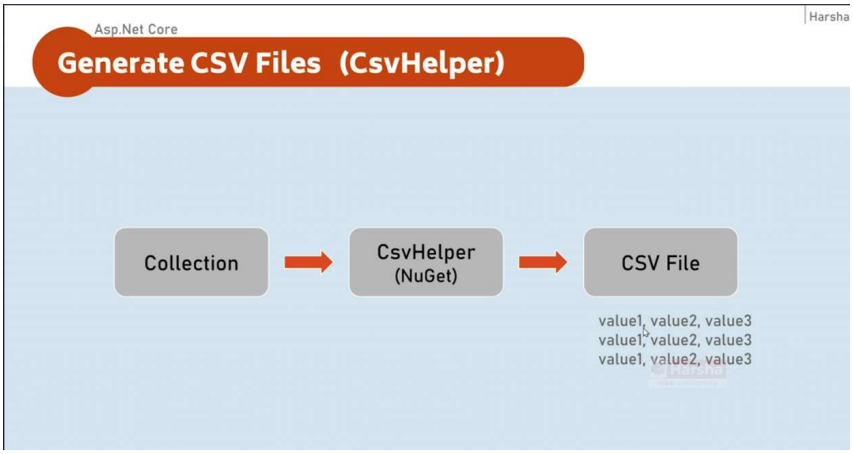



```
        [Route("PersonsPDF")]
        public async Task<IActionResult> PersonsPDF()
        {
            //Get list of persons
            List<PersonResponse> persons =  await _personsService.GetAllPersons();

            //Return view as pdf

            //by default, every controller action method contains some 'ViewData'. We have to supply the same.
            return new ViewAsPdf("PersonsPDF", persons, ViewData)
            {
                PageMargins = new Rotativa.AspNetCore.Options.Margins()
                {
                    Top = 20,
                    Right = 20,
                    Bottom = 20,
                    Left = 20
                },
                PageOrientation = Rotativa.AspNetCore.Options.Orientation.Landscape
            };
        }
    }
}
```

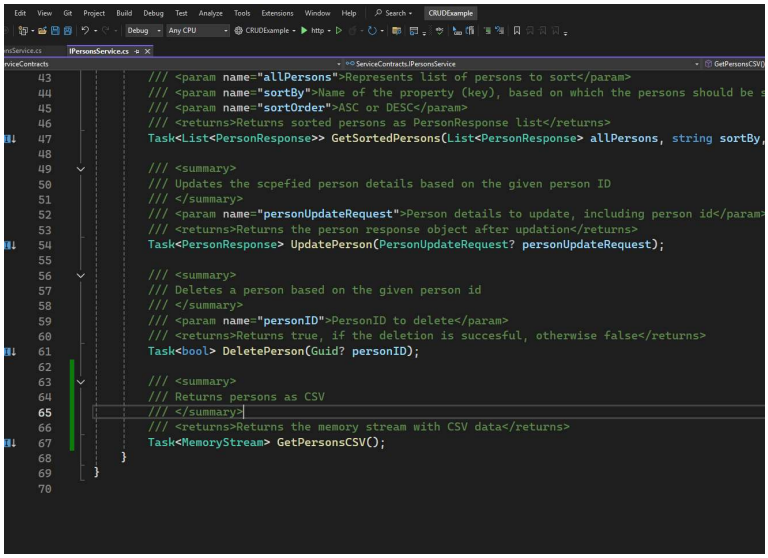



```
29
30      app.UseStaticFiles();
31      app.UseRouting();
32      app.MapControllers();
33
34      Rotativa.AspNetCore.RotativaConfiguration.Setup("wwwroot",wkhtmltopdfRelativePath:"Rotativa");
35
36      app.Run();
37
```

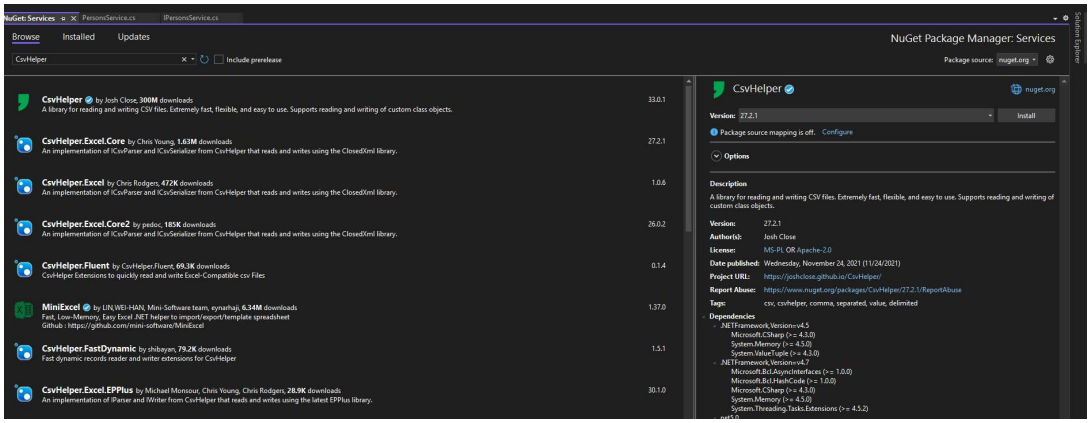


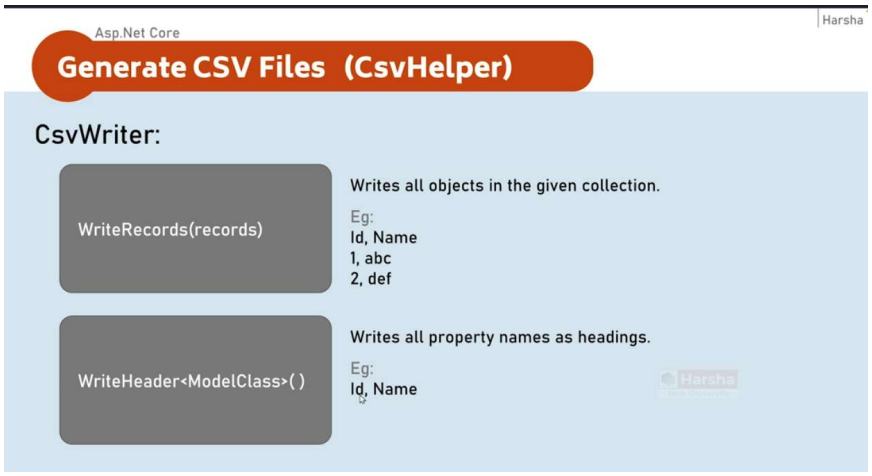


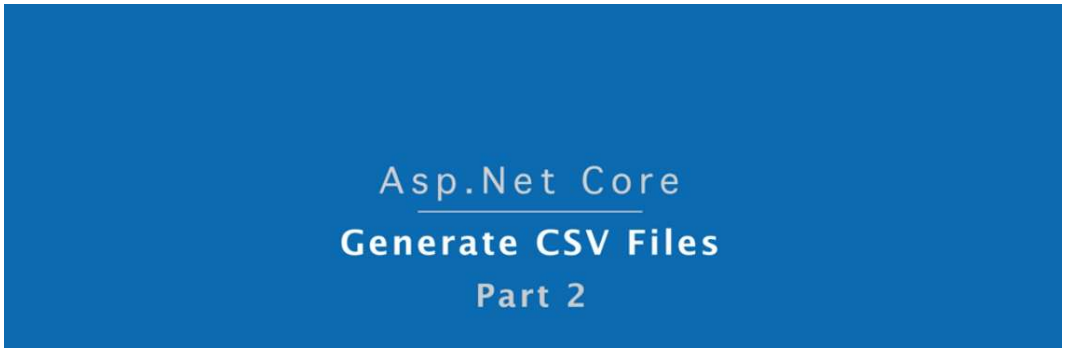
Now our next goal is to generate the data in the form of a CSV file (comma-separated values).
In the CSV file, each record is generated as one line, where a comma is the separator between the values.
For example, this is the person ID of the first person, then person name, and person email of the same person. Then, on the next row, the next record, and so on.
So, generally, for importing and exporting the data, you have to use the CSV file based on the client requirements.
For example, you can download Google contacts as a CSV file for migrating the same into another email provider.
So, in a similar way, assume our client requirement is that he is asking for a CSV file download for the persons.
So, we have to convert the person's collection into a CSV file by using a third-party package called **CsvHelper**.
Let me demonstrate the same.

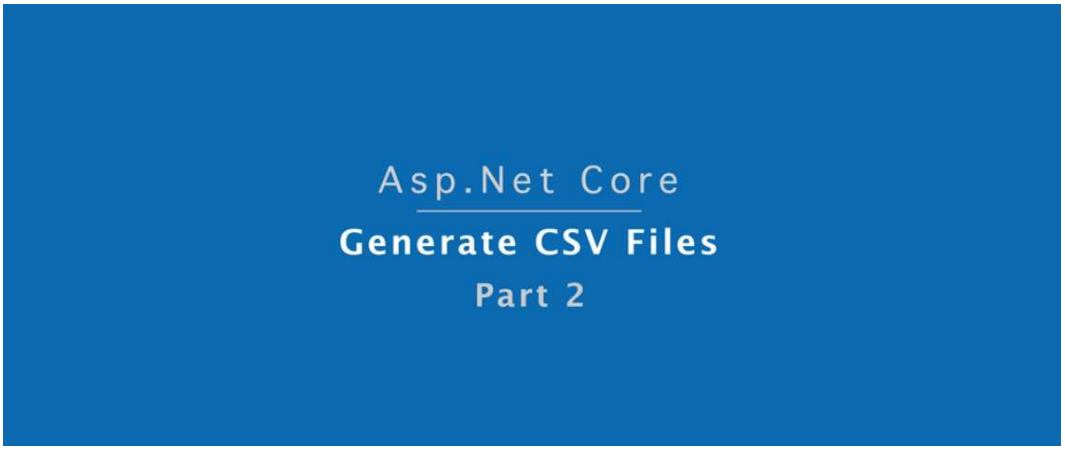


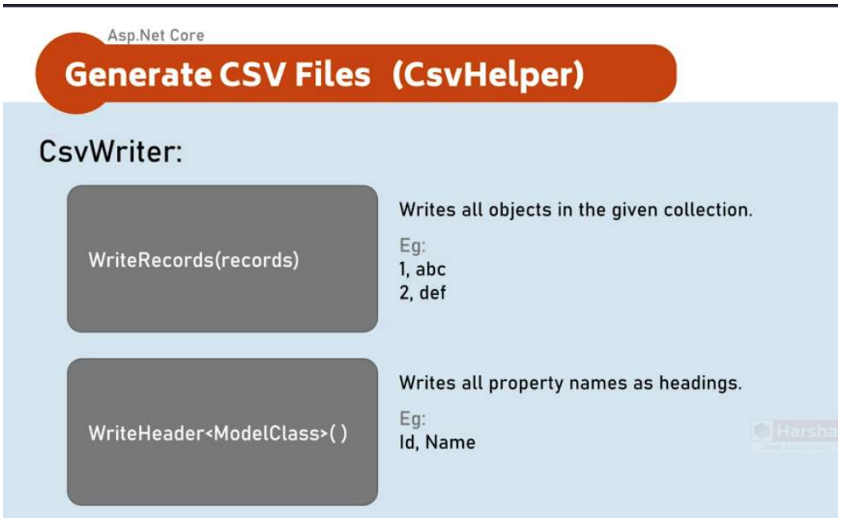
```
43          /// <param name="allPersons">Represents list of persons to sort</param>
44          /// <param name="sortBy">Name of the property (key), based on which the persons should be s
45          /// <param name="sortOrder">ASC or DESC</param>
46          /// <returns>Returns sorted persons as PersonResponse list</returns>
47          Task<List<PersonResponse>> GetSortedPersons(List<PersonResponse> allPersons, string sortBy,
48
49          /// <summary>
50          /// Updates the scpefied person details based on the given person ID
51          /// </summary>
52          /// <param name="personUpdateRequest">Person details to update, including person id</param>
53          /// <returns>Returns the person response object after updation</returns>
54          Task<PersonResponse> UpdatePerson(PersonUpdateRequest? personUpdateRequest);
55
56          /// <summary>
57          /// Deletes a person based on the given person id
58          /// </summary>
59          /// <param name="personID">PersonID to delete</param>
60          /// <returns>Returns true, if the deletion is succesful, otherwise false</returns>
61          Task<bool> DeletePerson(Guid? personID);
62
63          /// <summary>
64          /// Returns persons as CSV
65          /// </summary>
66          /// <returns>Returns the memory stream with CSV data</returns>
67          Task<MemoryStream> GetPersonsCSV();
68      }
69  }
70
```

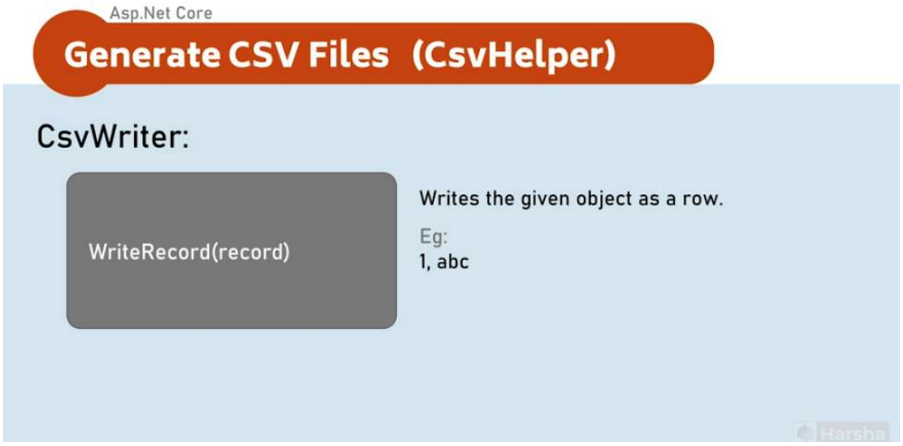




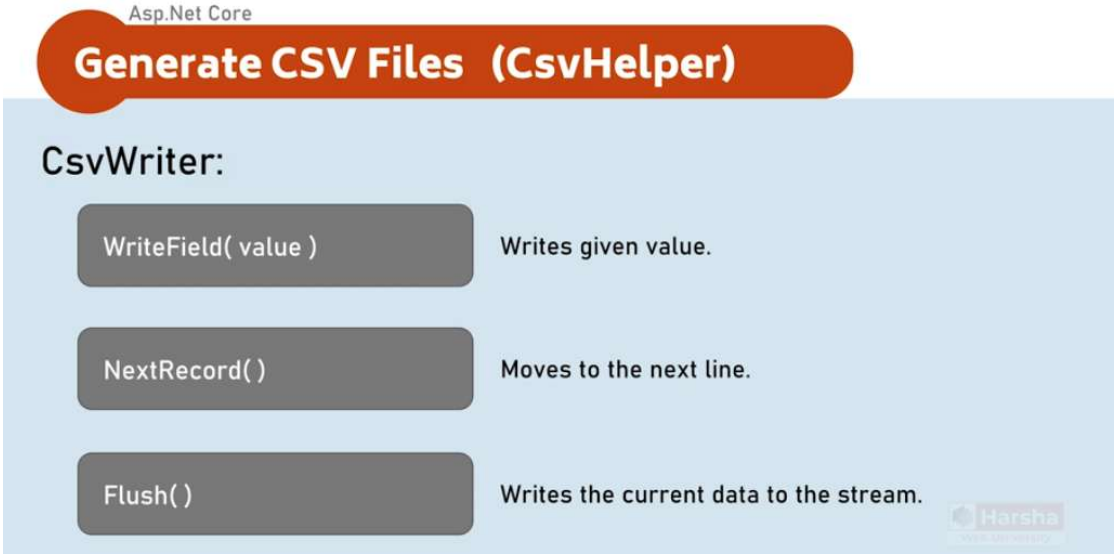






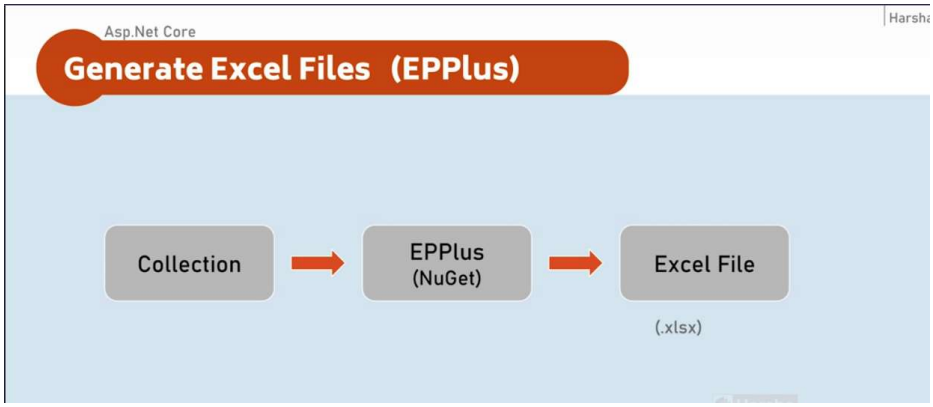


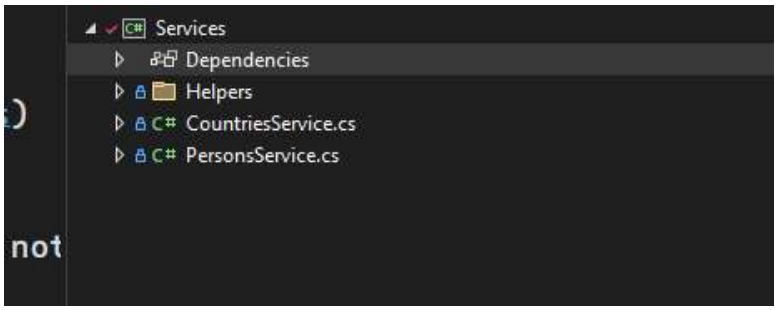
The problem with the **WriteRecords** and **WriteHeader** methods is that they don't let you choose which property values should be written to the stream.
By default, all the property values are written through this method.
And by default, all the property names (i.e., header names) are written through this method.
But if you want to **customize the values** or **change the order of values**, then it's better to use the **WriteField** method.
Actually, there is one more method called **WriteRecord**, which writes an entire single record, but we don't want to use that. Instead, we prefer **WriteField**.

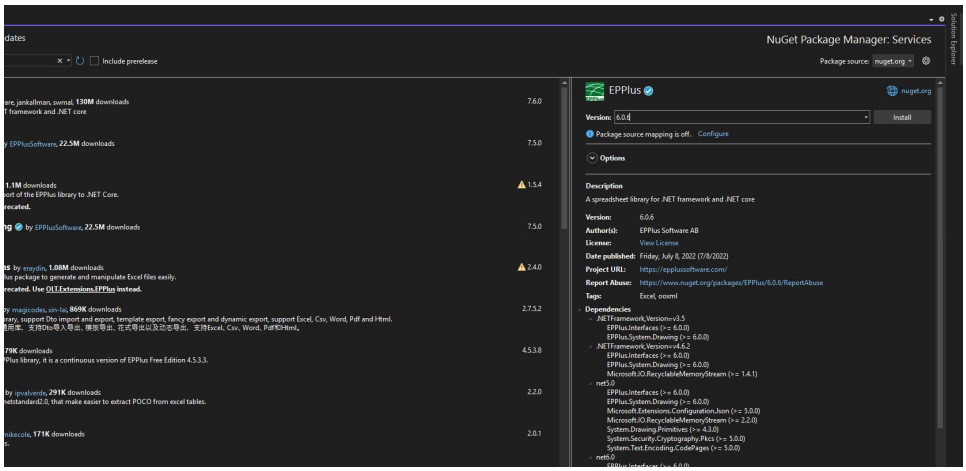


It writes a single value, no matter whether it is a value or a field heading.
When you want to move the cursor to the next line, you can use **NextRecord**.
For each record, we have to call the **Flush** method in order to write the data to the stream. Otherwise, the data will not be written to the stream.
Let me show that here.

Sometimes the client requirement will be to generate the data into an **Excel file**.
For example, there is a **Person's** table, and the client wants to download the person's information into an Excel file, just like a report or something.
For that case, you can use the third-party package called **EPPlus**, which helps convert the existing collection of data into an **Excel file** with a **.XLS** extension.
Let me show that.

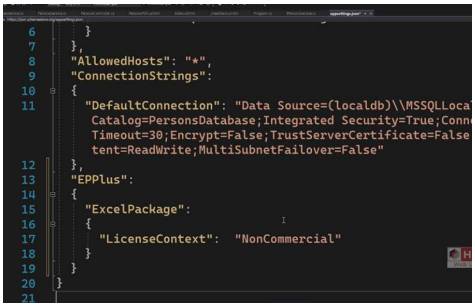

Asp.Net Core
Generate Excel Files (EPPlus)
Collection
EPPlus
(NuGet)
Excel File
(.xlsx)

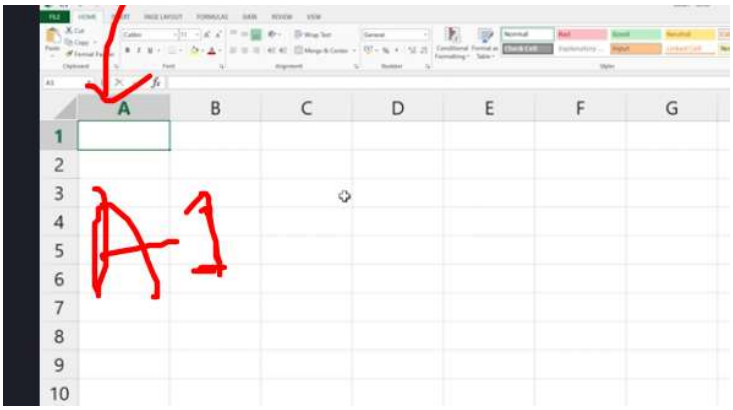

Services
Dependencies
Helpers
CountriesService.cs
PersonsService.cs

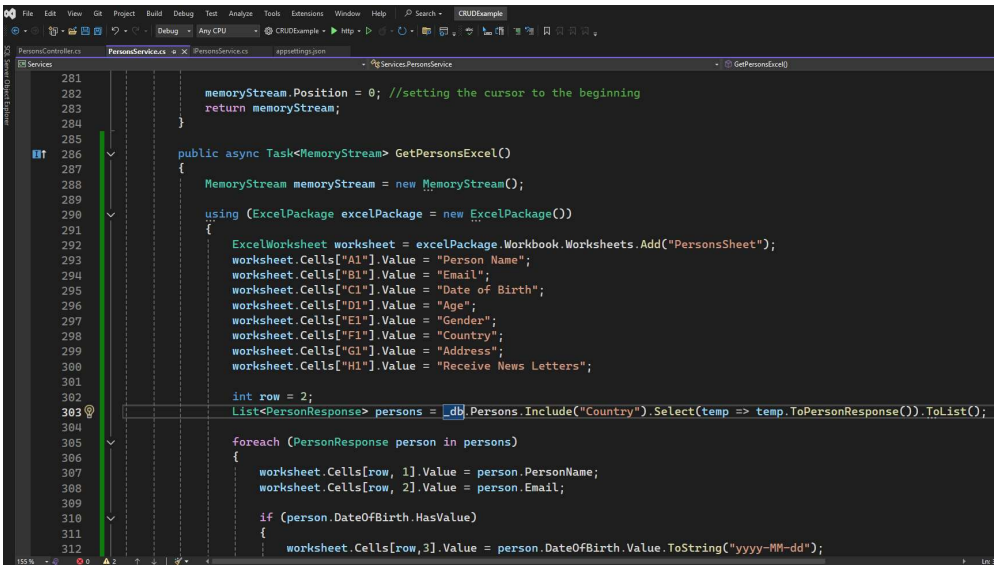

NuGet Package Manager: Services
EPPlus
Version:
Install
Description
A spreadsheet library for .NET framework and .NET core
Author(s):
EPPlus Software AB
Dependencies


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'CRUDExample' (5 of 5 projects)
CRUDExample
Connected Services
Dependencies
Properties
wwwroot
Controllers
Views
appsettings.json
countries.json
persons.json
Program.cs
CRUDTests

It automatically reads this.





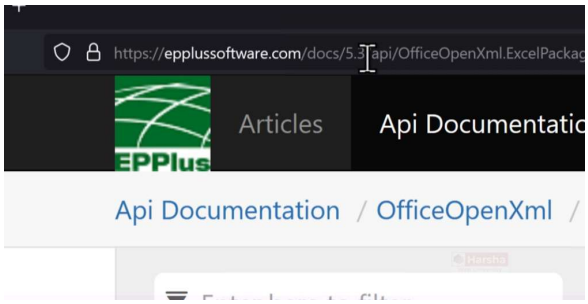


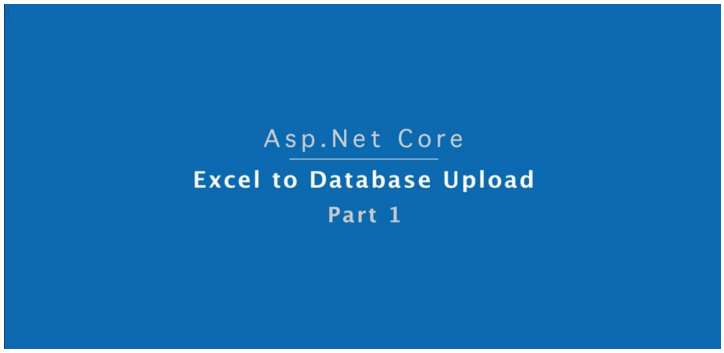
```
            memoryStream.Position = 0; //setting the cursor to the beginning
            return memoryStream;
        }

        public async Task<MemoryStream> GetPersonsExcel()
        {
            MemoryStream memoryStream = new MemoryStream();

            using (ExcelPackage excelPackage = new ExcelPackage())
            {
                ExcelWorksheet worksheet = excelPackage.Workbook.Worksheets.Add("PersonsSheet");
                worksheet.Cells["A1"].Value = "Person Name";
                worksheet.Cells["B1"].Value = "Email";
                worksheet.Cells["C1"].Value = "Date of Birth";
                worksheet.Cells["D1"].Value = "Age";
                worksheet.Cells["E1"].Value = "Gender";
                worksheet.Cells["F1"].Value = "Country";
                worksheet.Cells["G1"].Value = "Address";
                worksheet.Cells["H1"].Value = "Receive News Letters";

                int row = 2;
                List<PersonResponse> persons = _db.Persons.Include("Country").Select(temp => temp.ToPersonResponse()).ToList();

                foreach (PersonResponse person in persons)
                {
                    worksheet.Cells[row, 1].Value = person.PersonName;
                    worksheet.Cells[row, 2].Value = person.Email;

                    if (person.DateOfBirth.HasValue)
                    {
                        worksheet.Cells[row,3].Value = person.DateOfBirth.Value.ToString("yyyy-MM-dd");
```

```
        [Route("PersonsExcel")]
        public async Task<IActionResult> PersonsExcel()
        {
            MemoryStream memoryStream = await _personsService.GetPersonsExcel();

            return File(memoryStream, "application/vnd.openxmlformats-officedocument.spreadsheetml.sheet", "persons.xlsx");
        }
    }
}
```

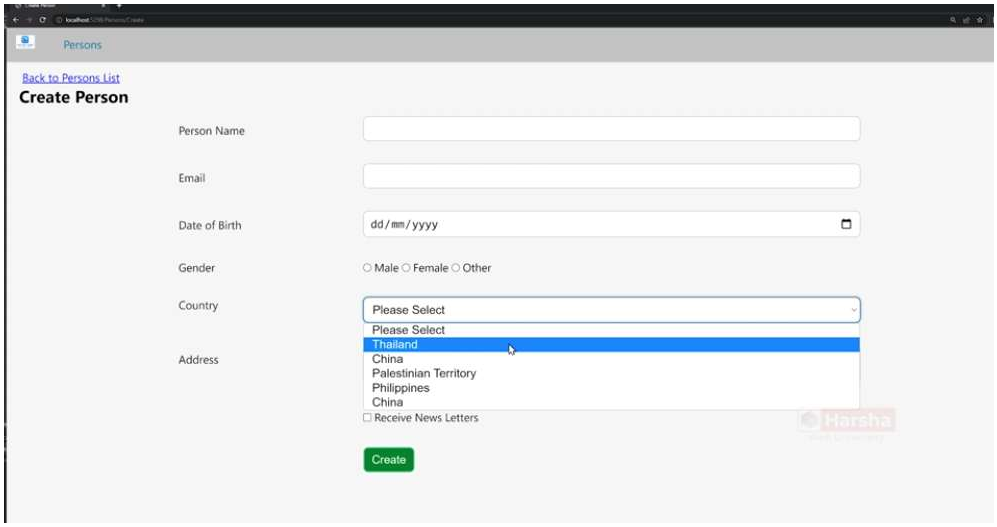
```
@model IEnumerable<PersonResponse>

@{
    ViewBag.Title = "Persons";
}
<form asp-controller="Persons" asp-action="Index" method="get">
    <h1>Persons</h1>

    <a asp-controller="Persons" asp-action="Create">Create Person</a>
    <a asp-controller="Persons" asp-action="PersonsPDF">Download as PDF</a>
    <a asp-controller="Persons" asp-action="PersonsCSV">Download as CSV</a>
    <a asp-controller="Persons" asp-action="PersonsExcel">Download as Excel</a>

    <div class="box flex">
        <div class="flex-1">
```

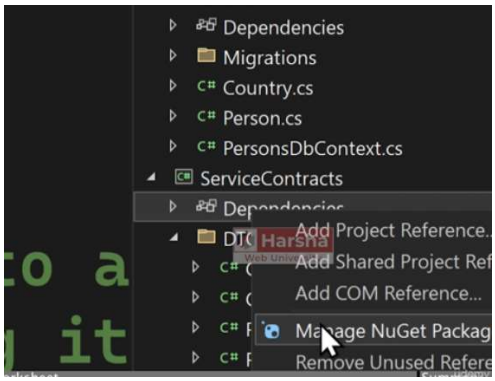
Official DOCS

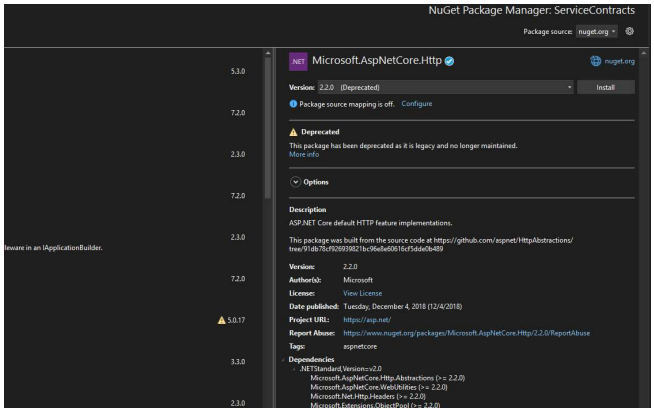


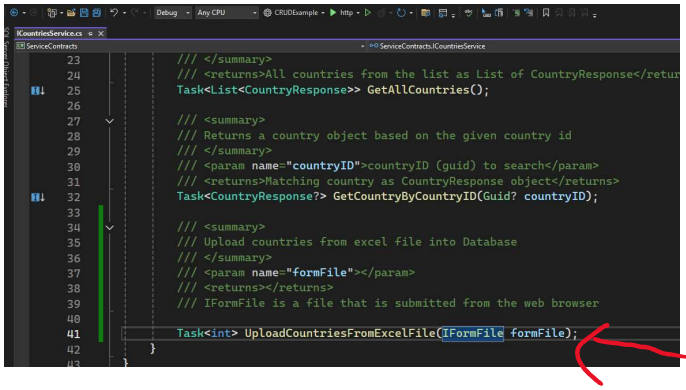


Now, I would like to enable the **upload functionality** for the database table.
Here, you will learn the concept of **file uploading** as well as **migrating data from an Excel file into the database**.
The requirement is:
For example, on the **Create Persons** page, you have a list of countries, right?
So, if the user wants to add more countries, instead of adding them **one by one** through an entry form, they can enter the list of countries in an **Excel file** and upload it to the database.
That's the requirement.
So, let’s see how to solve it.


Persons
Back to Persons List
Create Person
Person Name
Email
Date of Birth
dd/mm/yyyy
Gender
Male Female Other
Country
Please Select
Please Select
Thailand
China
Palestinian Territory
Philippines
China
Address
Receive News Letters
Create


Dependencies
Migrations
Country.cs
Person.cs
PersonsDbContext.cs
ServiceContracts
Add Project Reference...
Add Shared Project Ref
Add COM Reference...
Manage NuGet Packag
Remove Unused Refere


NuGet Package Manager: ServiceContracts
Microsoft.AspNetCore.Http
Version: 2.2.0 (Deprecated)
Install
Deprecated
This package has been deprecated as it is legacy and no longer maintained.
Options
Description
ASP.NET Core default HTTP feature implementations.
Version: 2.2.0
Author(s): Microsoft
License: View License
Date published: Tuesday, December 4, 2018 (12/4/2018)
Project URL: https://asp.net/
Tags: aspnetcore
Dependencies


ICountriesService.cs
ServiceContracts
ServiceContracts.ICountriesService
/// </summary>
/// <returns>All countries from the list as List of CountryResponse</retur
Task<List<CountryResponse>> GetAllCountries();
/// <summary>
/// Returns a country object based on the given country id
/// </summary>
/// <param name="countryID">countryID (guid) to search</param>
/// <returns>Matching country as CountryResponse object</returns>
Task<CountryResponse?> GetCountryByCountryID(Guid? countryID);
/// <summary>
/// Upload countries from excel file into Database
/// </summary>
/// <param name="formFile"></param>
/// <returns></returns>
/// IFormFile is a file that is submitted from the web browser
Task<int> UploadCountriesFromExcelFile(IFormFile formFile);

We have to create a template for the end user