1. Create a Console application **EFMigration**
2. Create a class `Blog.cs` and `Post.cs`:

```
public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }

    public virtual List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public virtual Blog Blog { get; set; }
}
```

You'll notice that we're making the two navigation properties (Blog.Posts and Post.Blog) virtual. This enables the Lazy Loading feature of Entity Framework. Lazy Loading means that the contents of these properties will be automatically loaded from the database when you try to access them.

3. From NuGet download **Microsoft.EntityFrameworkCore.SqlServer** Package

```
Install-Package Microsoft.EntityFrameworkCore.SqlServer
```

4. Add BloggingContext.cs:

```
using Microsoft.EntityFrameworkCore;

namespace EFMigration;

public class BloggingContext : DbContext
{
    protected override void OnConfiguring(
        DbContextOptionsBuilder optionsBuilder)
    {
        string connection = "Data Source=localhost\\SQLEXPRESS;" +
                            "Initial Catalog=Blog;" +
                            "Integrated Security=true;" +
                            "MultipleActiveResultSets=true;";
        optionsBuilder.UseSqlServer(connection);
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>().ToTable("Blog");
        modelBuilder.Entity<Post>().ToTable("Post");
    }
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }
}
```

5. Add DbInitializer.cs:

```
namespace EFMigration;
```

```
public class DbInitializer
{
    public static void Initialize(BloggingContext context)
    {
        context.Database.EnsureCreated();

        // Look for any students.
        if (context.Blogs.Any())
        {
            return;   // DB has been seeded
        }

        var blogs = new Blog[]
        {
            new Blog {Id = 1, Name = "MyBlog"}
        };
        foreach (var b in blogs)
        {
            context.Blogs.Add(b);
        }
        context.SaveChanges();

        var posts = new Post[]
        {
            new Post{ BlogId = 1, Title = "My first post", Content = "Hi! This is 
my very first post!" }
        };
        foreach (var p in posts)
        {
            context.Posts.Add(p);
        }
        context.SaveChanges();
    }
}
```

The `EnsureCreated` method is used to automatically create the database

The preceding code checks if the database exists:

- If the database is not found;
    - It is created and loaded with test data. It loads test data into arrays rather than List<T> collections to optimize performance
- If the database is found, it takes no action.

6. In Program.cs add:

```
using var db = new BloggingContext();

DbInitializer.Initialize(db);

// Create and save a new Blog
Console.Write("Enter a name for a new Blog: ");
var name = Console.ReadLine();

var blog = new Blog { Name = name };
db.Blogs.Add(blog);
db.SaveChanges();

// Display all Blogs from the database
```

```
    var query = from b in db.Blogs
        orderby b.Name
        select b;

    Console.WriteLine("All blogs in the database:");
    foreach (var item in query)
    {
        Console.WriteLine(item.Name);
    }

    Console.WriteLine("Press any key to exit...");
    Console.ReadKey();
```

Whenever the data model changes:

- Delete the database.
- Update the seed method, and start afresh with a new database.

Part 2

1. Add Nuget Package `Microsoft.EntityFrameworkCore.Design`
2. Check installed dotnet tools

```
C:\ dotnet tool list -g

Package Id      Version      Commands      Manifest
---------------------------------------------------
```

3. Update or install Install EF Core tools as a global tool

```
C:\ dotnet tool update dotnet-ef -g

Tool 'dotnet-ef' was successfully updated from version '6.0.6' to version
'6.0.7'.
```

Or

```
C:\ dotnet tool install --global dotnet-ef
```

4. Move in the csproj folder then Delete the database

```
C:\ cd C:\Users\natalil\repos\EFMigration\EFMigration
C:\ dotnet ef database drop
Build started...
Build succeeded.
Are you sure you want to drop the database 'Blog' on server
'localhost\SQLEXPRESS'? (y/N)
y
Dropping database 'Blog' on server 'localhost\SQLEXPRESS'.
Database 'Blog' did not exist, no action was taken.
```

Create an initial Migration

5. goto into csproj folder and enter this command:

```
C:\ cd C:\Users\natalil\repos\EFMigration\EFMigration
C:\ dotnet ef migrations add InitialCreate
Build started...
Build succeeded.
Done. To undo this action, use 'ef migrations remove'
```

Examine Up and Down methods

When you executed the migrations add command, EF generated the code that will create the database from scratch. This code is in the `Migrations` folder, in the file named `<timestamp>_InitialCreate.cs`

The `Up` method of the `InitialCreate` class creates the database tables that correspond to the data model entity sets, and the `Down` method deletes them, as shown in the following example.

Migrations calls the `Up` method to implement the data model changes for a migration

When you enter a command to roll back the update, Migrations calls the `Down` method.

This code is for the initial migration that was created when you entered the

`migrations add InitialCreate`

command

The migration name parameter ("`InitialCreate`" in the example) is used for the file name and can be whatever you want (i.e. “Pippo”).

It's best to choose a word or phrase that summarizes what is being done in the migration. For example, you might name a later migration "`AddDepartmentTable`".

If you created the initial migration when the database already exists, the database creation code is generated but it doesn't have to run because the database already matches the data model. When you deploy the app to another environment where the database doesn't exist yet, this code will run to create your database, so it's a good idea to test it first. That's why you dropped the database earlier -- so that migrations can create a new one from scratch.

The data model snapshot

Migrations creates a snapshot of the current database schema in `Migrations/SchoolContextModelSnapshot.cs.`

When you add a migration, EF determines what changed by comparing the data model to the snapshot file.

Use the `dotnet ef migrations remove` command to remove a migration

`dotnet ef migrations remove` deletes the migration and ensures the snapshot is correctly reset

If `dotnet ef migrations remove` fails, use `dotnet ef migrations remove -v` to get more information on the failure.

Apply the migration

6. At the moment the database does not exist.
   Enter the following command to create the database and tables in it.

```
   C:\ dotnet ef database update
Build started...
Build succeeded.
Applying migration '20220716164118_InitialCreate'.
Done.
```

7. NOTE: at the moment the database is still empty… RUN the application so thant `DbInitializer.Initialize(db);` seeds the data

Evolving your model

8. Now we need to add a *creation timestamp* to our posts. Our model looks like:

```
public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    public DateTime CreatedTimestamp { get; set; }

    public int BlogId { get; set; }
    public virtual Blog Blog { get; set; }
}
```

the model and the production database are now out of sync - we must add a new column to your database schema.

9. Let's create a new migration for this (from the project folder)

```
dotnet ef migrations add AddPostCreatedTimestamp

Build started...
Build succeeded.
Done. To undo this action, use 'ef migrations remove'
```

Note that we give migrations a descriptive name, to make it easier to understand the project history later.

Since this isn't the project's first migration, **EF Core now compares your updated model against a snapshot of the old model**, before the column was added; the model snapshot is one of the files generated by EF Core when you add a migration, and is checked into source control. **Based on that comparison, EF Core detects that a column has been added, and adds the appropriate migration.**

10. You can now apply your migration as before:

```
dotnet ef database update

Build started...
Build succeeded.
Applying migration '20220717100256_AddPostCreatedTimestamp'.
Done.
```

Excluding parts of your model

11. Sometimes you may want to reference types from another `DbContext`. This can lead to migration conflicts
    To prevent this, exclude the type from the migrations of one of the `DbContexts`

```
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<IdentityUser>()
            .ToTable("AspNetUsers", t => t.ExcludeFromMigrations());
    }
```

Other references:

1. Managing Migrations
   https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/managing?tabs=dotnet-core-cli
2. Applying Migrations
   https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/applying?tabs=dotnet-core-cli
3. Entity Framework Core tools reference
   https://docs.microsoft.com/en-us/ef/core/cli/