Entity Framework Core (EF Core) is an object-relational mapping (ORM) framework developed by Microsoft. It is a lightweight, extensible, and cross-platform version of Entity Framework, which enables developers to work with databases using .NET and .NET Core applications.

EF Core follows a code-first approach, which means that you define your database schema and relationships using C# or VB.NET classes, and EF Core takes care of creating the database and executing SQL queries based on those classes. This approach simplifies the development process by allowing you to focus on the application's business logic rather than writing complex SQL statements.

There are two primary approaches when working with EF Core:

Database-First Approach: In this approach, you start with an existing database schema and generate the corresponding entity classes and DbContext (database context) based on that schema. EF Core provides tools to reverse engineer the database schema into classes, allowing you to work with the database using object-oriented programming paradigms.

Code-First Approach: This approach involves defining the entity classes and their relationships in your code first, without an existing database. EF Core then generates the database schema and tables based on your classes. You can use attributes or fluent API configurations to specify additional details about the database schema, such as table names, column names, data types, indexes, etc. With the code-first approach, you have full control over the database schema and can easily update it as your application evolves.

Both approaches provide flexibility and enable you to map database entities to classes, perform CRUD operations (Create, Read, Update, Delete), define relationships, and execute complex queries using LINQ (Language Integrated Query). EF Core supports various database providers, including SQL Server, SQLite, PostgreSQL, MySQL, and more, allowing you to work with different databases seamlessly.

How to use EFCORE Code First Approach

Using the code-first approach in Entity Framework Core (EF Core) involves defining your entity classes and their relationships in code, and then letting EF Core generate the corresponding database schema based on those classes. Here's a step-by-step guide on how to use the code-first approach in EF Core:

Set up the Project: Start by creating a new ASP.NET Core project or any other .NET Core project in your preferred development environment.

Install EF Core Packages: Add the necessary NuGet packages to your project. The essential packages include Microsoft.EntityFrameworkCore and the database provider package you intend to use (e.g., Microsoft.EntityFrameworkCore.SqlServer for SQL Server). You can install these packages using the NuGet Package Manager or the .NET CLI.

Define Entity Classes: Create your entity classes that represent the tables in your database. Each class should correspond to a table, and properties within the class represent the table's columns. Decorate the properties with appropriate attributes to specify constraints, such as primary key, foreign key, required fields, etc.

Create the DbContext: Create a class that derives from DbContext. This class represents the database context and serves as the entry point for interacting with the database. In the DbContext class, define a DbSet property for each entity class you created. This property allows you to query and perform CRUD operations on the corresponding table.

Configure the DbContext: Override the OnConfiguring method in your DbContext class to specify the database provider and connection string. For example, if you're using SQL Server, you can use the following code:

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder.UseSqlServer("YourConnectionString");
  optionsBuilder.UseMySqlServer("YourConnectionString");

}
```

Enable Migrations: Open a command prompt or terminal, navigate to your project's root directory, and run the following command to enable migrations:

dotnet ef migrations add InitialCreate This command creates an initial migration that represents the current state of your entity classes and generates the necessary migration files.

Apply Migrations: After enabling migrations, apply the initial migration to create the database schema. Run the following command:

dotnet ef database update: This command applies the pending migrations to the database, creating the tables according to your entity classes.

Interact with the Database: Now that your database schema is created, you can use the DbContext to query, insert, update, and delete records from the database using LINQ or other EF Core query methods.

Modify the Entity Classes: As your application evolves and you need to make changes to the database schema, update your entity classes accordingly. Once you've made the changes, create a new migration and apply it to the database using the dotnet ef migrations add and dotnet ef database update commands.

By following these steps, you can effectively use the code-first approach in EF Core to define your entity classes and let EF Core handle the database schema creation and migration.

List of commonly used data annotations in ASP.NET Core

[Required]: Specifies that a property is required and must have a value.
[StringLength(maximumLength)]: Sets the maximum length for a string property.
[Range(minimum, maximum)]: Specifies the range of values allowed for a numeric property.
[RegularExpression(pattern)]: Validates that a property matches a specified regular expression pattern.
[EmailAddress]: Ensures that a string property contains a valid email address format.
[Phone]: Validates that a string property contains a valid phone number.
[Url]: Verifies that a string property is a valid URL.
[CreditCard]: Validates that a string property contains a valid credit card number.
[DataType(DataType.Date)]: Specifies the type of data expected for a property, such as a date, time, or string.
[Display(Name = "SomeName")]: Provides a friendly display name for a property.
[DisplayFormat(DataFormatString = "SomeFormat")]: Specifies the display format for a property, such as date or currency format.
[MaxLength(maximumLength)]:
Sets the maximum length for a string property.
[MinLength(minimumLength)]:
Sets the minimum length for a string property.
[Compare("OtherProperty")]:
Compares the value of a property with another property in the same model.
[RegularExpression(pattern, ErrorMessage = "SomeErrorMessage")]:
Specifies a custom error message when the regular expression pattern validation fails.
[DisplayFormat(ConvertEmptyStringToNull = true)]:
Converts an empty string value to null when binding the property.
[Key]: Indicates that a property is the primary key for the entity.
[Column("ColumnName")]: Specifies the database column name for a property.

[ForeignKey("OtherEntity")]: Defines a foreign key relationship between entities.
[NotMapped]: Excludes a property from being mapped to the database.

List of commonly used data migration commands in Entity Framework Core (EF Core):

dotnet ef migrations add [MigrationName]:
Creates a new migration with the specified name. This command generates migration files in the project's Migrations folder, representing the changes to the database schema.

dotnet ef migrations script:
Generates a SQL script of the migrations that haven't been applied to the database yet. This script can be used to inspect the SQL statements that will be executed when applying the migrations.

dotnet ef database update:
Applies any pending migrations to the database, updating the database schema to the latest migration. If no migrations have been applied yet, it creates the database based on the initial migration.

dotnet ef database update [MigrationName]:
Applies a specific migration to the database. Use this command to update the database schema to a specific migration without applying all pending migrations.

dotnet ef database update 0:
Reverts all applied migrations and resets the database schema to the initial state defined by the initial migration.

dotnet ef migrations remove:
Removes the last applied migration. This command rolls back the database schema changes made by the last migration.

dotnet ef migrations list:
Lists all available migrations, showing their names, status (applied or pending), and the date they were created.

dotnet ef migrations script [FromMigration] [ToMigration]:
Generates a SQL script for migrations between two specific migration points. This command allows you to generate a script that updates the database schema from one migration to another, without applying the migrations.

dotnet ef migrations remove [MigrationName]:

Removes a specific migration from the project. Use this command if you want to delete a migration that hasn't been applied yet.

The dotnet ef command-line tool provides various options and additional commands for managing database migrations. You can run these commands from the command prompt or terminal within your project's root directory.

Some commonly used properties available in Fluent API for configuring entity models in ASP.NET Core:

HasKey: Specifies the primary key property(s) for an entity.

HasAlternateKey: Defines an alternate key property(s) for an entity.

HasIndex: Configures an index on one or more properties of an entity.

Property: Configures the properties of an entity, including their data type, nullability, maximum length, precision, and more.

IsRequired: Specifies that a property is required and cannot be null.

HasMaxLength: Sets the maximum length of a string property.

HasDefaultValue: Specifies the default value for a property.

IsUnicode: Determines whether a string property should be Unicode or non-Unicode.

HasColumnType: Sets the database column data type for a property.

HasPrecision: Sets the precision and scale of a decimal property.

HasOne: Configures a one-to-one relationship with another entity.

WithMany: Configures a one-to-many or many-to-many relationship with another entity.

HasForeignKey: Specifies the foreign key property(s) for a relationship.

OnDelete: Configures the delete behavior of a relationship.

Ignore: Excludes a property from being mapped to the database.

ToTable: Specifies the table name for an entity.

ToView: Configures an entity to be mapped to a database view.

HasQueryFilter: Applies a global query filter to an entity to restrict the results returned from the database.

IsConcurrencyToken: Marks a property as a concurrency token for optimistic concurrency control.

ValueGeneratedOnAdd/ValueGeneratedOnAddOrUpdate/ValueGeneratedNever: Specifies how a property value is generated by the database, such as for identity columns.

These are just a few examples of the properties available in Fluent API for configuring entity models in ASP.NET Core. Fluent API provides a powerful and flexible way to customize the behavior and mapping of your entity classes to the database schema.

Here's an example of how to use data seeding in Entity Framework Core to populate the database with initial data:

Define the Seed Data: Create a class to represent your seed data. This class should contain the initial data you want to populate in the database. For example:

```
public static class SeedData
{
    public static void SeedUsers(ModelBuilder modelBuilder)
    {
        if (!modelBuilder.Entity<User>().Any(a => a.UserId == 1))
        {
            modelBuilder.Entity<User>().HasData(
                new User{ User= 1, Name = "John Smith" }
            );
        }
    }
}
```

Configure Data Seeding: In your DbContext class, override the OnModelCreating method and call the seed methods to populate the initial data. For example:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    // Other configurations...
```

```
    SeedData.SeedUsers(modelBuilder);
 }
```

Apply the Migrations: Apply the migrations to create the database schema if you haven't done so already. Run the following command:

dotnet ef database update

Perform Data Seeding: To perform the data seeding, you can override the OnModelCreating method in your DbContext class and call the SaveChanges method after adding the seed data. For example:

```
public class ApplicationDbContext : DbContext
{
    // DbContext code...

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        // Other configurations...

        SeedData.SeedUsers(modelBuilder);
         base.OnModelCreating(modelBuilder);
    }

    public override int SaveChanges()
    {
        ChangeTracker.DetectChanges();

        // Save changes and apply data seeding
        var entities = ChangeTracker.Entries()
            .Where(e => e.State == EntityState.Added || e.State ==
EntityState.Modified)
            .Select(e => e.Entity);

        foreach (var entity in entities)
        {
            // Perform any necessary data modifications

            // Example: Set created/modified dates
            if (entity is BaseEntity baseEntity)
            {
                if (baseEntity.CreatedDate == default)
                    baseEntity.CreatedDate = DateTime.UtcNow;

                baseEntity.ModifiedDate = DateTime.UtcNow;
            }
        }
```

```
        return base.SaveChanges();
    }
}
```
In this example, the SeedData class contains seed methods: SeedUsers. These methods use the HasData method to specify the initial data for the User.

In the OnModelCreating method of the DbContext, you call the seed methods to populate the initial data when the database schema is created.

To ensure the data seeding occurs whenever changes are saved to the database, you override the SaveChanges method in the DbContext. In this method, you retrieve the entities that are being added or modified and perform any necessary modifications before saving the changes.

By following this approach, the initial data will be seeded into the database when the migrations are applied, and any subsequent changes to the database will trigger the data seeding process.