Entity Framework (EF) and the SQL table structure

Entity Framework (EF) does not have to exactly match the SQL table structure; however, the class does need to map properly to the database table so that EF can handle the data correctly.

Basic Mapping: By default, EF will expect that your class properties map one-to-one to the columns in the SQL table. This includes the same names (case-insensitive) and compatible data types. If your class doesn't match the table exactly, EF may not know how to map the properties or handle the data.

Flexible Mapping with Attributes: You can use data annotations (attributes) to map a class property to a different column name, ignore certain properties, or configure other mappings.

Sample

```
[Table("MyTable")]
public class MyClass
{
  [Key]
  public int Id { get; set; }

  [Column("DifferentColumnName")]
  public string MyProperty { get; set; }

  [NotMapped]
  public string IgnoredProperty { get; set; }
}
```

Fluent API: If not using attributes then configure the mappings in the DbContext class using the Fluent API.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  modelBuilder.Entity<MyClass>()
    .ToTable("MyTable")
    .Property(p => p.MyProperty)
    .HasColumnName("DifferentColumnName");
}
```

Partial Matching: Your class does not need to have properties for every column in the SQL table. EF will only insert data for the properties that are mapped. However, if a required column in the database is not included in your class and you attempt to insert a record, you'll get an error (such as missing column values).

While the class doesn't have to match the table exactly, you should ensure that it is properly mapped using attributes or Fluent API and any necessary columns are present for successful insert operations.

No Primary Key: EF core entities require a primary key unless you explicitly define the entity as keyless. If you need to represent a view, raw SQL query, or some data that doesn't have a primary key then configure it as a keyless entity. What this means is that the EF Core model can contain keyless entity types which can be used to carry out database queries against data that doesn't contain key values.

By default, all entities in EF Core require a primary key unless you explicitly define the entity as keyless.

If you want to define your DBContext as a keyless entity

```
public class MyDbContext : DbContext
{
  public DbSet<OptOutEvent> OptOutEvents { get; set; }

  protected override void OnModelCreating(ModelBuilder modelBuilder)
  {
    modelBuilder.Entity<OptOutEvent>().HasNoKey();
  }
}
```

This is commonly used for querying data that doesn't need to be tracked for updates or doesn't have a unique identifier, such as views or reports.

A little bit more on this…

```
using Microsoft.EntityFrameworkCore;

public class OptOutEvent
{
  public int Id { get; set; } // primary key
  public string EventType { get; set; }
  public DateTime EventDate { get; set; }
  // other properties...
}

public class MyDbContext : DbContext
{
  public DbSet<OptOutEvent> OptOutEvents { get; set; } // for OptOutEvent

  protected override void OnModelCreating(ModelBuilder modelBuilder)
  {
    // fluent API configurations
    modelBuilder.Entity<OptOutEvent>().HasKey(e => e.Id); // config primy key
  }

  protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
  {
    // define database connection string
    optionsBuilder.UseSqlServer("YourConnectionStringHere");
  }
}
```