Entity Framework Core

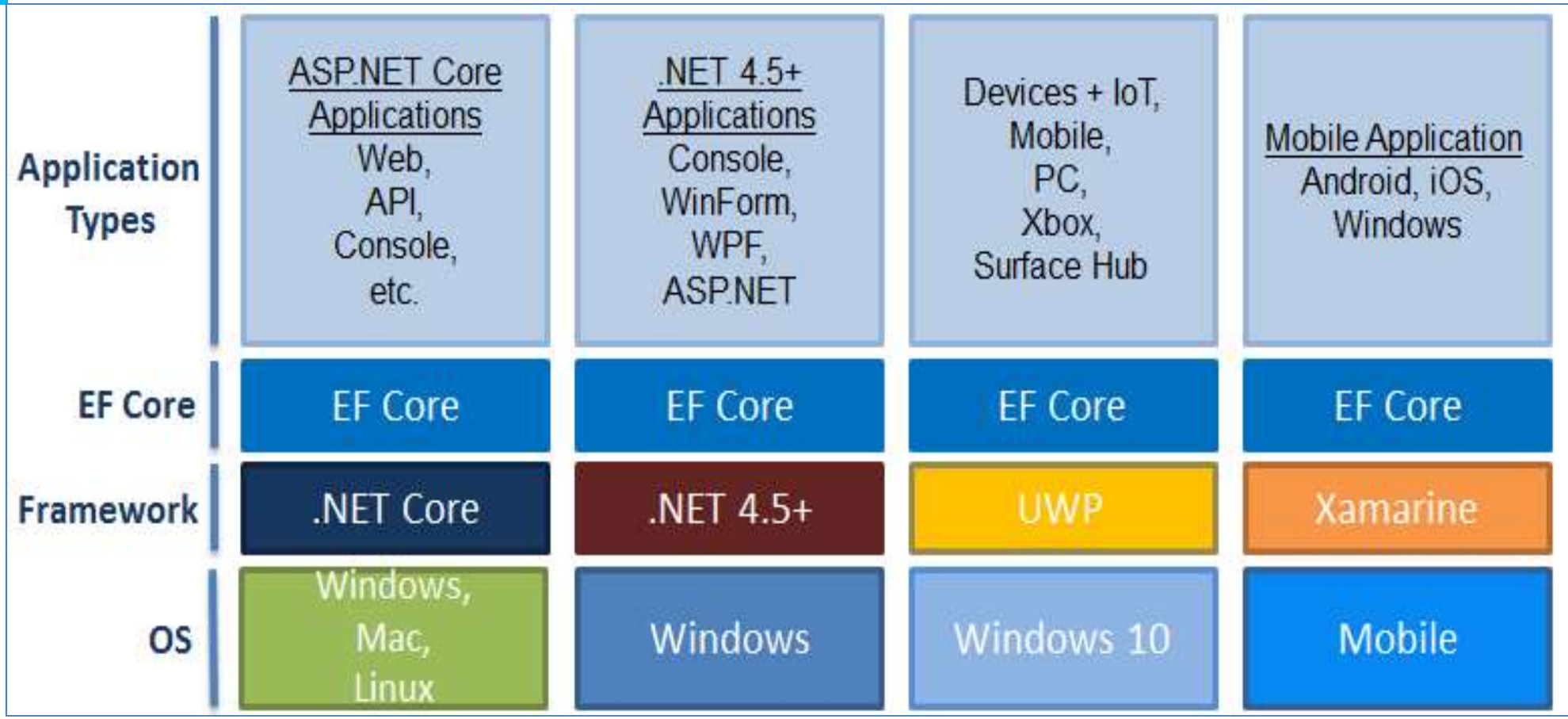
Presented by: Mandaar Jani
Authored by: Mandaar Jani

Agenda – Session 1

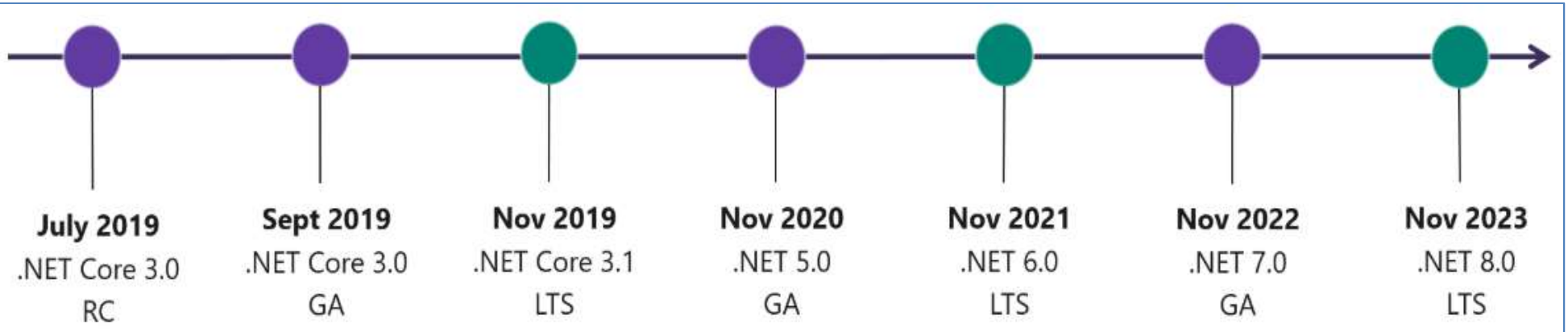
- .NET & EF Core Overview
- Entity Framework vs EF Core
- .NET & EF Core Timelines, Features, CLI
- CLI Quick Demos
- Object Relational Mappers (ORMs)
- Entity Framework Core
- Demo - Database First Approach
- Demo - Code First Approach

.NET – A Unified Platform

EF Core Stack Position



.NET Core Timeline



- .NET Core 3.0 release in September
- .NET Core 3.1 = Long Term Support (LTS)
- .NET 5.0 release in November 2020
- Major releases every year, LTS for even numbered releases
- Predictable schedule, minor releases if needed

EF Core Timeline

Stable releases

Release	Target framework	Supported until	Links
EF Core 3.1	.NET Standard 2.0	December 3, 2022 (LTS)	Announcement
EF Core 3.0	.NET Standard 2.1	March 3, 2020	Announcement / Breaking changes
~~EF Core 2.2~~	.NET Standard 2.0	Expired December 23, 2019	Announcement
EF Core 2.1	.NET Standard 2.0	August 21, 2021 (LTS)	Announcement

.NET Core – Quick look

- Program Class
  - Application start point to create Host for the application, configure and start instance of application.
- Startup Class
  - Configure Service method → To configure application's services using *IServiceCollection*
  - Configure method → To create application's request pipeline using *IApplicationBuilder*
- Dependency Injection
  - Built-in IoC Container (For both Framework Provided and Custom Services)

    → supports constructor & method injection.
  - Service Lifetimes → Singleton, Transient, Scoped
  - RequestServices collection → To get service instance without injection

Features

- .NET Core Open Source - https://github.com/dotnet/core
- EF Core Open Source - https://github.com/dotnet/efcore
- Cross Platform - Hosting Agents – Windows / Ubuntu / Mac OS
- Modular – Package driven
- Command Line – dotnet new , restore, build , test, run, publish
- Hosting – Program & Startup Class
- Built-in IoC Container, Dependency Injection, Middleware, Configuration
- Logging – Audit (Audit.WebApi) and Error logging (Nlog)

Object Relational Mappers

Software	Platform	Availability	License	Version
Dapper	.NET 4.0	Open source	Apache License 2.0	1.8 NuGet
Django	Python	Open source	BSD licenses	2.1 (1 August 2018)
Entity Framework Core	.net core	Open Source	Apache License 2.0	2.0
Hibernate	Java Virtual Machine	Open source	GNU Lesser General Public License	4.2.5 / August 28, 2013
ADO.NET Entity Framework	.NET 4.5	Part of .NET 4.5	Apache License 2.0[3]	v6.0 (2014)
nHibernate	.NET 4.5	Open source	GNU Lesser General Public License	4.0 (2014-08-17[4])

Object Relational Mappers

- Database Abstraction
    - It abstracts away the database system so that switching from MySQL to PostgreSQL, or whatever flavor you prefer, is easy-peasy.
- Use Native developer friendly language for both DDL & DML
    - You get to write in the language you are already using anyway.
    - Fluent language code understood to all developers – do not need SQL expertise
- Built-in Features
    - Leverage numerous features like Transactions, Migrations, Connection Pooling, Seeds, Streams and other goodies
- Optimized Query Compilation
- Written, Tried and tested by experts to compile into SQL queries in more standardized way than written by different developers by different way.

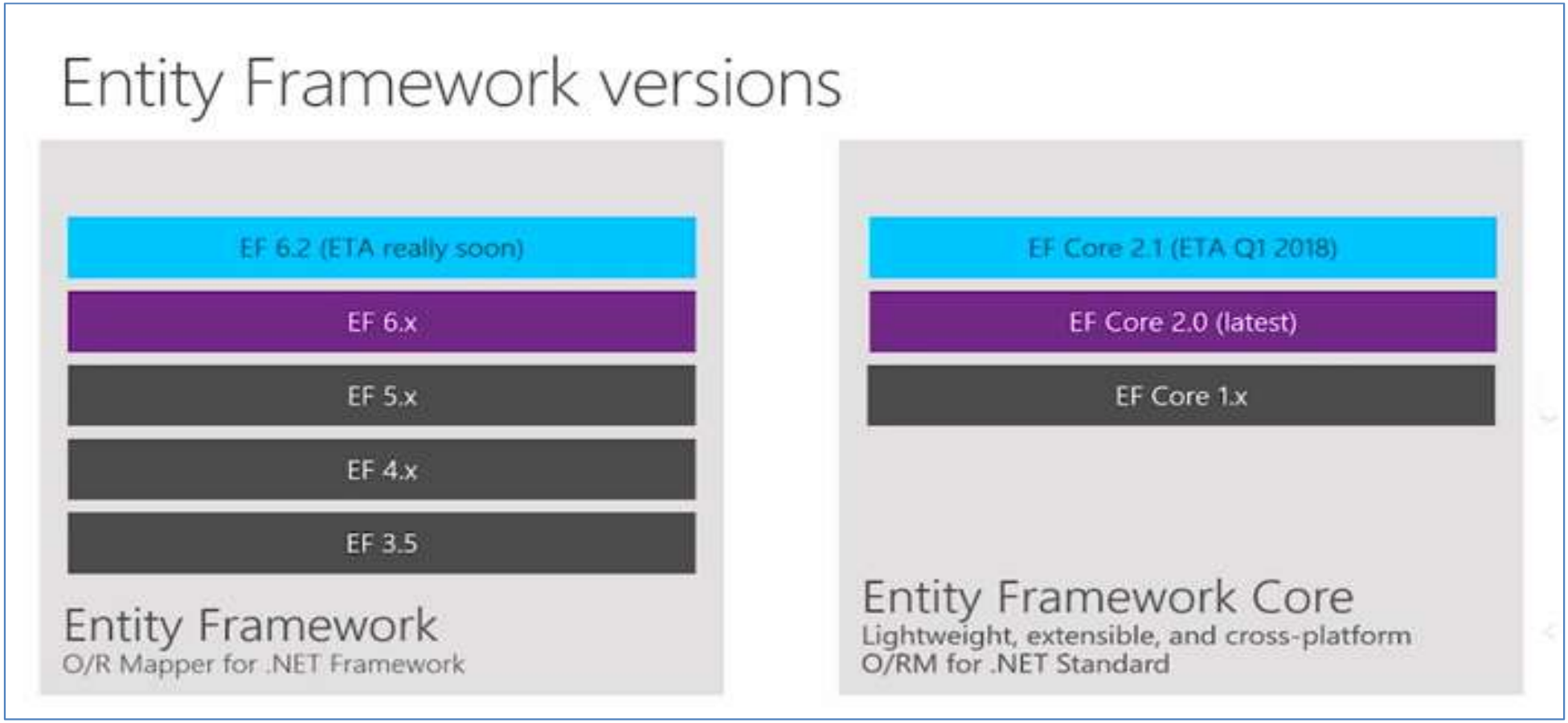
Micro ORM

- A subset of ORM tools , example – Dapper –can be used for smaller scale projects or smaller scale data layer
- Eliminates the need for heavy lifting of major ORM functionalities. You don't need it, don't add it.
- Limitations
  - Caching – is not supported
  - Relationships – No one to one, many to many relationships supported - loading an object doesn't load all related objects automatically – for this, construct the queries on need basis.
  - No designer – mostly don't have it
  - Migration – Most of ORM tools support some kind of migration or code first approach, where you design your model and after executing application database objects are created based on model - use 3rd party tools on top of it.
  - Large scale application – Not recommended
  - LTS applications – Not recommended

Entity Framework Core

Let's Dive into it

Entity Framework vs EF Core



Features

- Entity Framework Core is a lightweight, extensible, open source and cross-platform version of populate EF data access (DAAB) technology
- EF Core can serve as an ORM , enabling .NET developers to work with a database using .NET objects and eliminating the need for most of the data-access code they usually need to write
- Supports SQL Server / SQL Azure, SQLite, Azure Cosmos DB, MySQL, PostgreSQL and other data providers.
- Porting from EF 6 to EF Core is possible whether or not we used EDMX-based model earlier
- Current version EF Core 3.x -> Next version EF Core 5.x to align with .NET 5.0
- LINQ overhaul – Single SQL statement per LINQ query
- Reverse engineering of database views – Keyless Entity Types
- C# 8.0 support
- Cosmos DB Support

Database First Approach – Demo - CLI

- dotnet ef –version
- dotnet ef dbcontext scaffold "Data Source=MANDAARJ\SQL2017;Initial Catalog=webapi;User ID=sa;Password=cybage@123;" Microsoft.EntityFrameworkCore.SqlServer
- Pipes –
- --tables or -t
- --force or -f
- --output-dir or –o
- --context or –c
- From Visual Studio -> Tools -> PackageManager Console –
- Scaffold-DbContext "Server=.\;Database=AdventureWorksLT2012;Trusted_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Model -Context "AdventureContext" -DataAnnotations

Code First Approach – Demo - CLI

- dotnet ef –version
- dotnet ef migrations add CreateDatabase
- dotnet ef database update
- Make any changes to the code model
- dotnet ef database update

EF Core Data Providers

- Microsoft.EntityFrameworkCore.SqlServer
- Microsoft.EntityFrameworkCore.Sqlite
- Microsoft.EntityFrameworkCore.InMemory
- Microsoft.EntityFrameworkCore.Cosmos
- Npgsql.EntityFrameworkCore.PostgreSQL
- Pomelo.EntityFrameworkCore.MySql
- Devart.Data.MySql.Efcore
- Devart.Data.Oracle.EFCore
- And many more available at as nugget packages

Agenda – Session 2

- Recap on Session 1
- Entity Framework Core
  - EF Core Conventions
  - DB Context
  - Querying & Saving Data
  - Using Stored Procedures
- Fluent API
- EF Power Tools

EF Core Conventions

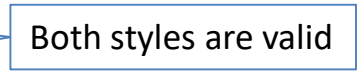
- Conventions are a set of rules hard-baked into EF core that govern how the model will be mapped to a database schema.
- There are ways to generate custom conventions by using some EF core's configuration options using attributes or Fluent API. But, all of these are static so for cases like dynamic properties Fluent API helps resolve them using chained methods.
- Primary Key, Foreign Key, FK Shadow Properties, Backing Fields, Table
- Schema – EF Core will map objects to the dbo schema by default.
- Columns – EF Core will map entity properties to DB columns with same name.
- Data Types – String properties are unlimited in size and mapped as nvarchar(max). Custom data types are mapped by creating a model with properties covering all types.
- Indexes – EF Core will always create indexes for foreign key and alternate keys.

EF Core Conventions – Primary Key

- If a property is named ID or <entity name>ID (not case-sensitive), it will be configured as primary key.
- EF core prefers ID over <entity name>ID in the event a class contains both such properties.
- EF core will specify that the PK column values are generated automatically by the database.

```
public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
}
```

```
public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }
}
```



EF Core Conventions – Foreign Key

- The convention for a foreign key is that it must have the same data type as principal entity's primary key property and the name must follow one of these patterns:
  - <navigation property name><principal primary key property name>Id
  - <principal class name><primary key property name>Id
  - <principal primary key property name>Id
- As with primary keys, FK property name matching is not case sensitive. EF core infers the multiplicity of a relationship from the nullability of the foreign key. If the property is not nullable, the relationship is registered as required.
- Order of precedence for the example mentioned:
- WriterAuthorId
- AuthorAuthorId
- AuthorId

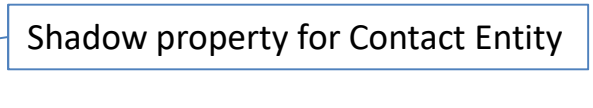
```
public class Author
{
    public int AuthorId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public ICollection<Book> Books { get; set; }
}
public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public Author Writer { get; set; }
    public string Isbn { get; set; }
}
```

EF Core Conventions – Foreign Key Shadow Properties

- If you choose not to explicitly include a foreign key property in the dependent end of the relationship. EF Core will create a shadow property using the pattern <principal primary key property name>Id.
- Shadow properties are extended properties that do not feature as part of the entity class but can be included in the model and are mapped to the database columns. Typically defined in OnModelCreating method.

```
public class SampleContext : DbContext
{
    public DbSet<Contact> Contacts { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Contact>()
            .Property<DateTime>("LastUpdated");
    }
}
public class Contact
{
    public int ContactId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
}
```



EF Core Conventions – Backing Fields

- Backing fields are used to preserve encapsulation.
- Typically used for fields like “Status” which could be applied as read only on the entities and fetched from the database. EF Core will automatically map a backing field if its name matches a pattern like:
  - _<camel-cased property name> ex. _publisher
  - _<property name> ex. _Publisher
  - m_<camel-cased property name>
  - m_<property name>

```
public class Book
{
    private Publisher _publisher;
    private decimal m_recommendedRetailPrice;
    public int BookId { get; set; }
    public string Title { get; set; }
    public int PublisherId { get; set; }
    public Publisher Publisher {
        get { return _publisher; }
        set { _publisher = value; }
    }
    public decimal RecommendedRetailPrice {
        get { return m_recommendedRetailPrice; }
        set { m_recommendedRetailPrice = value; }
    }
}
```

EF Core Conventions – Table

- EF Core will map an entity to a table with the same name as its corresponding DbSet<TEntity> property.
- If it does not have corresponding DbSet<TEntity> , 2nd check is it will look for a fully defined relationship.
- Books table will be created because of DbSet<Book>
- Publisher table will be created because of fully defined relationship with Books entity.

```
public class LibraryContext : DbContext
{
    public DbSet<Book> Books { get; set; }
}
public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public int PublisherId { get; set; }
    public Publisher Publisher { get; set; }
}
public class Publisher
{
    public int PublisherId { get; set; }
    public string Name { get; set; }
    public ICollection<Book> Books { get; set; }
}
```

EF Core DbContext

- EF Core DbContext class represents a session with a database and provides an API for communicating with DB

  with following capabilities:
  - **Database Connections** – is responsible for opening and managing connections to DB.
  - **Data operations** such as querying and persistence – CRUD operations directly on entities thereby table.
  - **Change Tracking** – sets the EntityState of an object according to the type of operation that the DB is asked for perform on it.
  - **Model building** – builds a conceptual model based on convention and configuration and maps to the DB.
  - **Data Mapping** – includes a data mapper layer responsible for mapping the results of SQL queries to entity instances and other types defined by the client application.
  - **Object Caching** – first level cache present if subsequent requests for the same object is made.
  - **Transaction management** – SaveChanges method uses a transactional approach wrapped in it as a single Unit of Work to apply pending changes to the DB and if an error occurs, it rolls back leaving the DB in an unmodified condition.

CRUD operations

- Basic CRUD operations could be done easily with LINQ to SQL type of queries.
- EF Core allows a NoTracking attribute to disallow unnecessary in memory tracking, here are a few samples

```
public async Task<IActionResult> Index()
{
    using (var context = new EFCoreWebDemoContext())
    {
        var model = await context.Authors.AsNoTracking().ToListAsync();
        return View(model);
    }

}
```

Viewing flat list of data

Create data one at a time

```
[HttpPost]
public async Task<IActionResult> Create([Bind("FirstName, LastName")] Author author)
{
    using (var context = new EFCoreWebDemoContext())
    {
        context.Add(author);
        await context.SaveChangesAsync();
        return RedirectToAction("Index");
    }
}
```

CRUD operations – Related data

- Include() method allows fetching related data with / without NoTracking attribute
- SaveChangesAsync() method allows saving single or multiple changes

```
public async Task<IActionResult> Index()
{
    using (var context = new EFCoreWebDemoContext())
    {
        var model = await context.Authors.Include(a => a.Books).AsNoTracking().ToListAs

        return View(model);
    }

}
```

Viewing relational data

Create relational data

```
public async Task<IActionResult> Create([Bind("Title, AuthorId")] Book book)
{
    using (var context = new EFCoreWebDemoContext())
    {
        context.Books.Add(book);
        await context.SaveChangesAsync();
        return RedirectToAction("Index");
    }
}
```

CRUD operations – Stored Procedures

- Mapping to DbQuery – DbQuery type was introduced in EF Core 2.1. It is a property of DbContext that acts similar to DbSet providing a root of LINQ queries.
- It does not allow to write to a database & only maps to a table or a view.
- However, there are additional ways using Fluent API, where we can build custom types and use them as retrieval mapping objects when querying a database or inserting into a database.
- Mapping to views can be done using DbQuery - Example
- Bonus –
  - Use Repository pattern to handle execution of Stored Procedures
  - Use object <-> object mapping for fetching data from and saving data to data layers
  - Build data service layers (mostly interface driven) to also mock the service layers in unit test cases

CRUD operations – Stored Procedures

```
public async Task<IActionResult> Create([Bind("FirstName, LastName")] Author author)
{
    using (EFCoreWebDemoContext context = new EFCoreWebDemoContext())
    {
        //context.Add(author);
        SqlParameter[] parameters = new SqlParameter[] {
        new SqlParameter { ParameterName = "@AuthorId", SqlDbType = SqlDbType.Int, Direction = ParameterDirection.Output},
        new SqlParameter { ParameterName = "@FirstName", SqlDbType = SqlDbType.VarChar, Size = 50, Direction = ParameterDirection.Input, SqlValue = author.FirstName},
        new SqlParameter { ParameterName = "@LastName", SqlDbType = SqlDbType.VarChar, Size = 75, Direction = ParameterDirection.Input, SqlValue = author.LastName}
        };
        try
        {
            context.Database.ExecuteSqlRaw("Exec AuthorSave @AuthorId OUT, @FirstName, @LastName", parameters);
        }
        catch (System.Exception e)
        {
            //log error
            ModelState.TryAddModelError("AddingAuthor", e.Message);
        }
        object returnvalue = parameters[0].Value;
        if (returnvalue != null && int.TryParse(returnvalue.ToString(), out int AuthorId) && AuthorId > 0)
        {
            //Log success message;
        }
        //await context.SaveChangesAsync();
        return RedirectToAction("Index");
    }
}
```

EF Core – Fluent API

- Fluent API provides methods for configuring various aspects of your model:
  - Model-wide Configuration
  - Type Configuration
  - Property Configuration
- Configurations are applied using Microsoft.EntityFrameworkCore.ModelBuilder class using an OnModelCreating method.

```
// series of statements
modelBuilder.Entity<Order>().Property(t => t.OrderDate).IsRequired();
modelBuilder.Entity<Order>().Property(t => t.OrderDate).HasColumnType("Date");
modelBuilder.Entity<Order>().Property(t => t.OrderDate).HasDefaultValueSql("GetDate()");
// fluent api chained calls
modelBuilder.Entity<Order>()
    .Property(t => t.OrderDate)
        .IsRequired()
        .HasColumnType("Date")
        .HasDefaultValueSql("GetDate()");
```

EF Core – Fluent API – Configuration classes

- Fluent API can also be written using separate configuration classes

```
public class OrderConfiguration : IEntityTypeConfiguration<Order>
{
    public void Configure(EntityTypeBuilder<Order> builder)
    {
        builder.HasKey(0 => o.OrderNumber);
        builder.Property(t => t.OrderDate)
            .IsRequired()
            .HasColumnType("Date")
            .HasDefaultValueSql("GetDate()"
    }
}
```

Key differences w.r.t EF 6

- Configuration class must implement IEntityTypeConfiguration<T> with type T
- Configuration is applied in a Configure method
- Configurations are added to ModelBuilder using ApplyConfiguration method

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.ApplyConfiguration(new OrderConfiguration());
}
```

Key differences w.r.t EF 6

- Configurations are added to ModelBuilder using ApplyConfiguration method

EF Core – Fluent API – Model Wide

- Schema – The default schema for EF core to create database objects is dbo. However, we can change the behavior using HasDefaultSchema method. No data annotations are available for this.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.HasDefaultSchema("MyCustomSchema");
}
```

- Exclude Type – Model wide configurations will require sometimes to ignore models to be created for composite types or DbQuery Types for execution of stored procedures for example. This can be done using Ignore<T> method
- Snapshot – Migrations creates a snapshot of current database schema. When adding migration, EF Core determines what changed by comparing the data model to snapshot file.

EF Core – Fluent API – Model Wide

```
public class SampleContext : DbContext
{
    public DbSet<Contact> Contacts { get; set; }
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Ignore<AuditLog>();
    }
}
public class Contact
{
    public int Id { get; set; }
    public string FullName { get; set; }
    public string Email { get; set; }
    public AuditLog AuditLog { get; set; }
}
public class AuditLog
{
    public int EntityId { get; set; }
    public int UserId { get; set; }
    public DateTime Modified { get; set; }
}
```

EF Core – Fluent API – Type Configuration

- Entity – EF Core provides a range of options for configuring types (entities) using Fluent API's ModelBuilder.Entity() method. It has a range of available methods -

Method Name	Usage
HasAlternateKey	Generates a unique constraint for the specified property or properties
HasAnnotation	Provides a means to apply annotations via the Fluent API
HasBaseType	Specifies the base type of the entity
HasIndex	Generates an index on the specified property or properties
HasKey	Denotes the specified property as the entity key
HasMany	Specifies the Many end of a relationship
HasOne	Specifies the One end of a relationship
Ignore |Denotes that the entity should be omitted from mapping	
ToTable	Specifies the database table that the entity should be mapped to
Property	Provides access to property configuration

EF Core – Fluent API – Property Configuration

- Property – EF Core provides a range of options for configuring types (entities) using Fluent API's EntityTypeBuilder.Property() method. It has a range of available methods -

Method Name	Usage
HasAnnotation	Provides a means to apply annotations via the Fluent API
HasColumnName	Specifies the name of the database column that the property should map to
HasColumnType	Specifies the data type of the database column that the property should map to
HasDefaultValue	Configures the default value of the database column that the property maps to
HasDefaultValueSql	Configures the default value expression for the database column that the property maps to
HasMaxLength	Specifies maximum length of data that can be stored for strings or binary data (arrays)
IsConcurrencyToken	Denotes that the property takes part in concurrency management
IsRequired	Configures the database column as not nullable
ValueGeneratedNever	Specifies that the database should not automatically generate values for the property
ValueGeneratedOnAdd	Specifies that values should only be generated automatically when new data is added
ValueGeneratedOnAddOrUpdate	Specifies that values should be generated automatically when data is added or updated

EF Core – Fluent API – Type Configuration

```
modelBuilder.Entity("MVC.Models.Book", b =>
    {
        b.Property<int>("BookId")
            .ValueGeneratedOnAdd()
            .HasColumnType("int")
            .HasAnnotation("SqlServer:ValueGenerationStrategy",
            SqlServerValueGenerationStrategy.IdentityColumn);

        b.Property<int>("AuthorId")
            .HasColumnType("int");

        b.Property<string>("Title")
            .HasColumnType("nvarchar(255)")
            .HasMaxLength(255);

        b.HasKey("BookId");

        b.HasIndex("AuthorId");

        b.ToTable("Books");
    });
```

Fluent API – Relationships – One-to-One

- EF Core currently, cannot determine the dependent entity in the relationship.
- The Has/With pattern is used to close the loop and fully define a relationship, the HasOne method is chained with the WithOne method.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Author>()
        .HasOne(a => a.Biography)
        .WithOne(b => b.Author)
        .HasForeignKey<AuthorBiography>(b => b.AuthorRef);
}
```

Fluent API – Relationships – One-to-Many

- The Has/With pattern is used to close the loop and fully define a relationship, the HasOne / HasMany method is chained with the WithOne / WithMany method.
- A company has many employees, each with one company.

```
protected override void OnModelCreating(Modelbuilder modelBuilder)
{
    modelBuilder.Entity<Company>()
        .HasMany(c => c.Employees)
        .WithOne(e => e.Company);
}
```

OR

```
protected override void OnModelCreating(Modelbuilder modelBuilder)
{
    modelBuilder.Entity<Employee>()
        .HasOne(e => e.Company)
        .WithMany(c => c.Employees);
}
```

Fluent API – Relationships – Many-to-Many

- With this kind of relation ships, we basically would want to create a join table or a juncture table.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<BookCategory>()
        .HasKey(bc => new { bc.BookId, bc.CategoryId });
    modelBuilder.Entity<BookCategory>()
        .HasOne(bc => bc.Book)
        .WithMany(b => b.BookCategories)
        .HasForeignKey(bc => bc.BookId);
    modelBuilder.Entity<BookCategory>()
        .HasOne(bc => bc.Category)
        .WithMany(c => c.BookCategories)
        .HasForeignKey(bc => bc.CategoryId);
}
```

```
public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public Author Author { get; set; }
    public ICollection<BookCategory> BookCategories { get; set; }
}
public class Category
{
    public int CategoryId { get; set; }
    public string CategoryName { get; set; }
    public ICollection<BookCategory> BookCategories { get; set; }
}
public class BookCategory
{
    public int BookId { get; set; }
    public Book Book { get; set; }
    public int CategoryId { get; set; }
    public Category Category { get; set; }
}
```

References

- https://www.learnentityframeworkcore.com/
- https://docs.microsoft.com/en-us/ef/core/
- .NET Core Open Source - https://github.com/dotnet/core
- EF Core Open Source - https://github.com/dotnet/efcore
- ORM –
- https://knexjs.org/ - JavaScript based ORM
- https://en.wikipedia.org/wiki/List_of_object-relational_mapping_software
- https://dapper-tutorial.net/dapper
- Database providers list - https://docs.microsoft.com/en-us/ef/core/providers/?tabs=dotnet-core-cli
- Shoot your questions at –

  Mandaar Jani – mandaarj@cybage.com