# DB and ORM

## References

1. Github link
2. Website or youtube link

## Database

### Relational Database

#### Join

Combines data or rows from two or more tables based on a common field between them. Related to each other using foreign key constraints.

***SELECT*** *column\_list*

***FROM*** *table1*

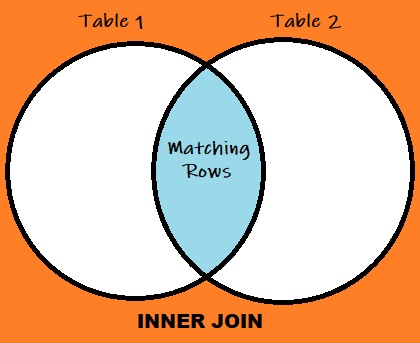
***INNER JOIN/LEFT JOIN/RIGHT JOIN/FULL JOIN*** *table2*

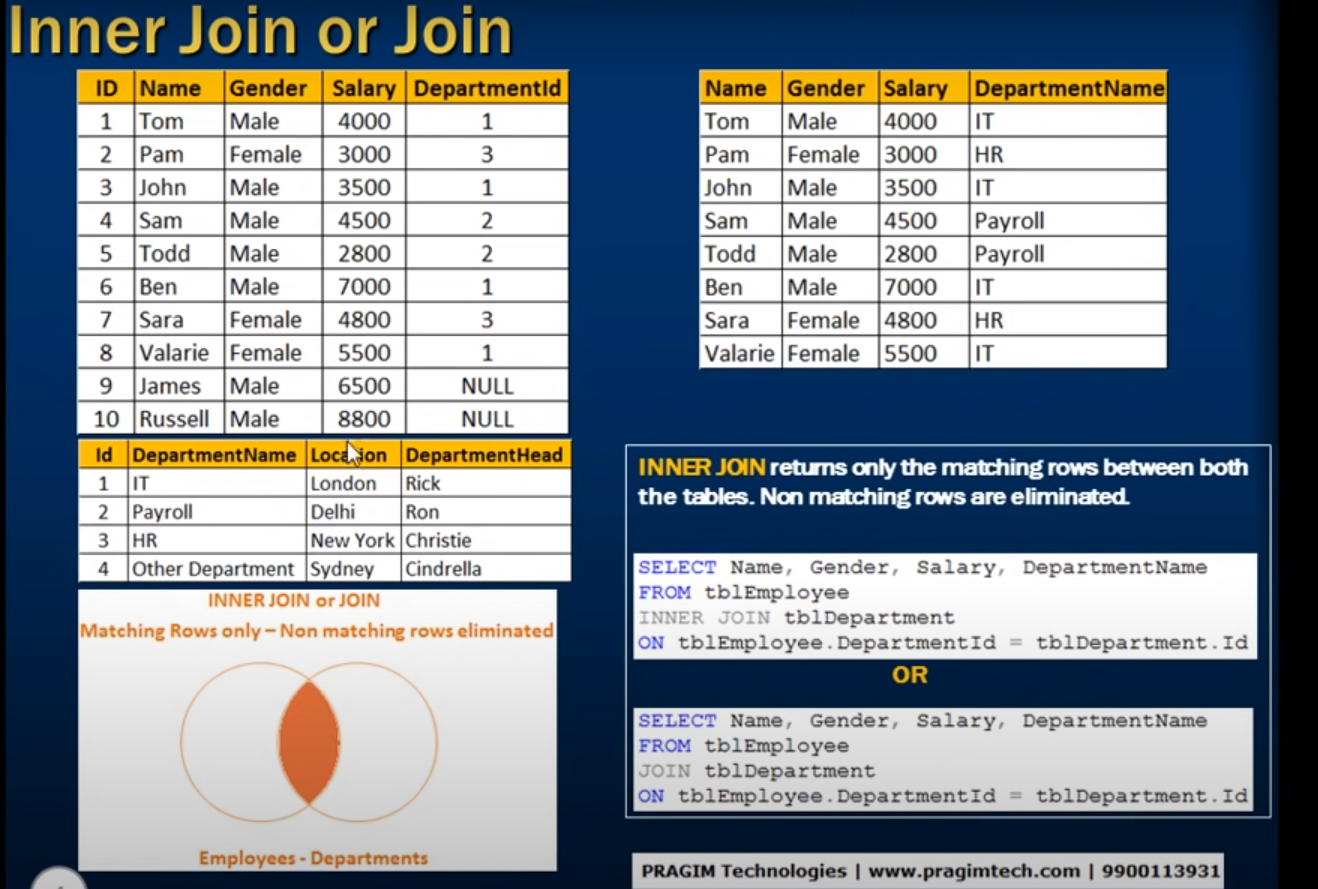
***ON*** *table1.column\_name = table2.column\_name;*

*Use* ***where*** *keyword to include additional conditions like* ***null or not null***

##### Inner Join

Matching rows only, non matching rows are eliminated

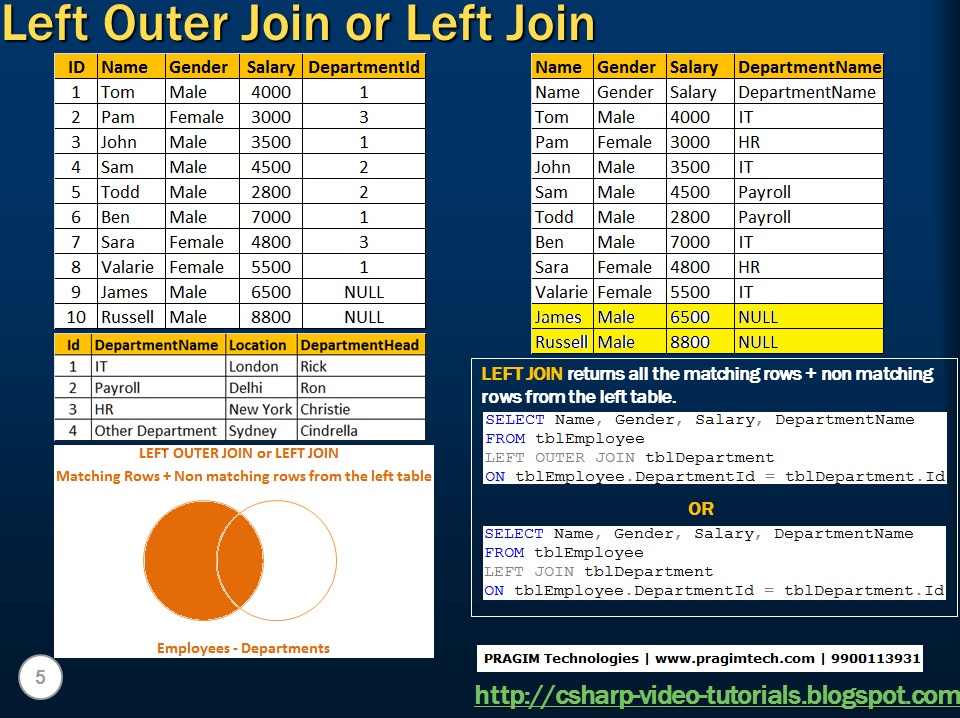




##### Outer Join

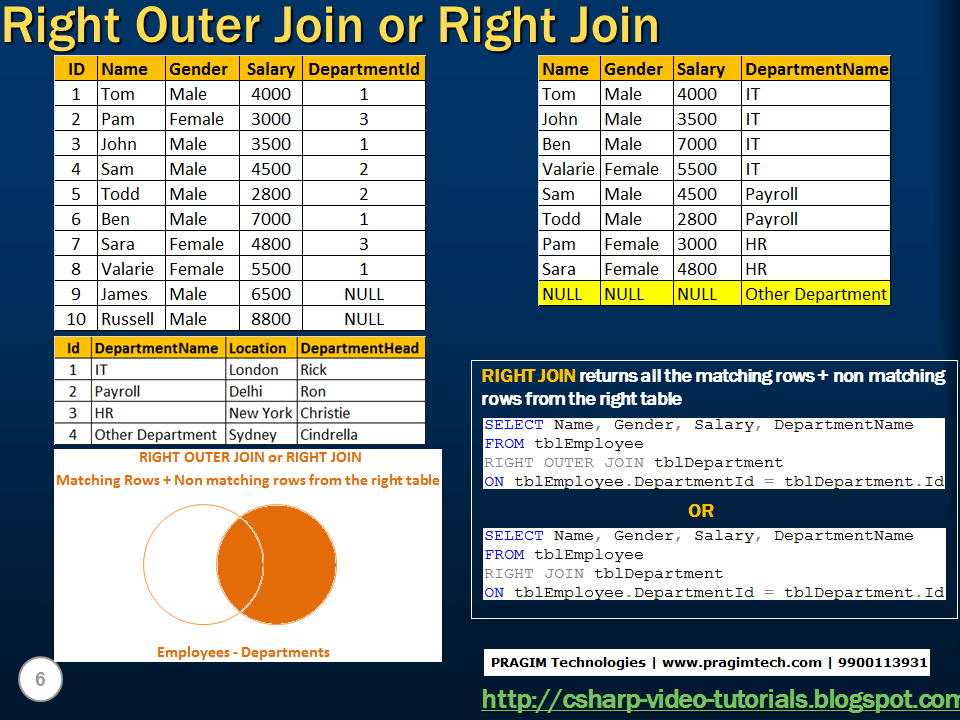
###### Left

All the matching rows + non matching rows from the left table



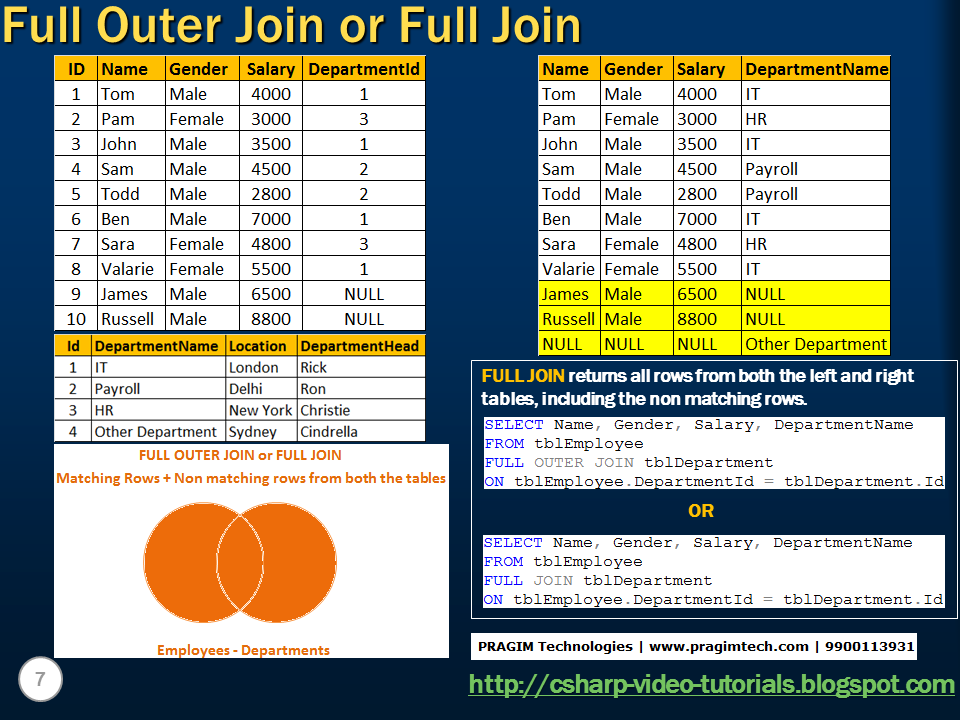
###### Right

All the matching rows + non matching rows from the right table



###### Full

All rows from left and right including non matching rows



##### Cross Join

Cartesian product of two tables involved in the join



##### Hash Join

### IDbConnection Interface

Represents an open connection to a data source, and is implemented by .NET data providers that access relational databases.

**Properties**

|  |  |
| --- | --- |
| **[ConnectionString](https://learn.microsoft.com/en-us/dotnet/api/system.data.idbconnection.connectionstring?view=net-7.0" \l "system-data-idbconnection-connectionstring)** | Gets or sets the string used to open a database. |
| **[ConnectionTimeout](https://learn.microsoft.com/en-us/dotnet/api/system.data.idbconnection.connectiontimeout?view=net-7.0" \l "system-data-idbconnection-connectiontimeout)** | Gets the time to wait (in seconds) while trying to establish a connection before terminating the attempt and generating an error. |
| **[Database](https://learn.microsoft.com/en-us/dotnet/api/system.data.idbconnection.database?view=net-7.0" \l "system-data-idbconnection-database)** | Gets the name of the current database or the database to be used after a connection is opened. |
| **[State](https://learn.microsoft.com/en-us/dotnet/api/system.data.idbconnection.state?view=net-7.0" \l "system-data-idbconnection-state)** | Gets the current state of the connection. |

**Methods**

|  |  |
| --- | --- |
| **[BeginTransaction()](https://learn.microsoft.com/en-us/dotnet/api/system.data.idbconnection.begintransaction?view=net-7.0" \l "system-data-idbconnection-begintransaction)** | Begins a database transaction. |
| **[BeginTransaction(IsolationLevel)](https://learn.microsoft.com/en-us/dotnet/api/system.data.idbconnection.begintransaction?view=net-7.0" \l "system-data-idbconnection-begintransaction(system-data-isolationlevel))** | Begins a database transaction with the specified [IsolationLevel](https://learn.microsoft.com/en-us/dotnet/api/system.data.isolationlevel?view=net-7.0) value. |
| **[ChangeDatabase(String)](https://learn.microsoft.com/en-us/dotnet/api/system.data.idbconnection.changedatabase?view=net-7.0" \l "system-data-idbconnection-changedatabase(system-string))** | Changes the current database for an open Connection object. |
| **[Close()](https://learn.microsoft.com/en-us/dotnet/api/system.data.idbconnection.close?view=net-7.0" \l "system-data-idbconnection-close)** | Closes the connection to the database. |
| **[CreateCommand()](https://learn.microsoft.com/en-us/dotnet/api/system.data.idbconnection.createcommand?view=net-7.0" \l "system-data-idbconnection-createcommand)** | Creates and returns a Command object associated with the connection. |
| **[Dispose()](https://learn.microsoft.com/en-us/dotnet/api/system.idisposable.dispose?view=net-7.0" \l "system-idisposable-dispose)** | Performs application-defined tasks associated with freeing, releasing, or resetting unmanaged resources.  (Inherited from [IDisposable](https://learn.microsoft.com/en-us/dotnet/api/system.idisposable?view=net-7.0)) |
| **[Open()](https://learn.microsoft.com/en-us/dotnet/api/system.data.idbconnection.open?view=net-7.0" \l "system-data-idbconnection-open)** | Opens a database connection with the settings specified by the ConnectionString property of the provider-specific Connection object. |

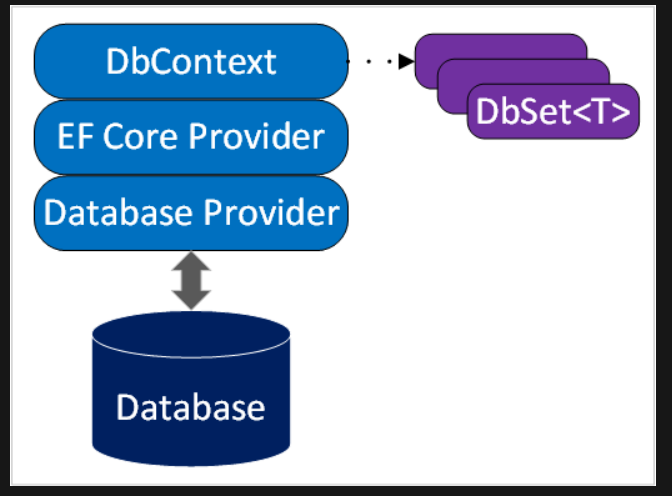
### Connection string

## Dapper - Lightweight

Identity will be used to set the value for particular column in increment seed value provided by user.

Sample code in github

## EF Core Architecture



### Concepts

Entity Framework Core is a popular object-relational mapping (ORM) framework for .NET applications. It provides a set of tools and libraries to work with relational databases using object-oriented programming paradigms. Here are some main concepts in Entity Framework Core:

DbContext: DbContext is a core class in Entity Framework Core that represents a session with the database. It is responsible for managing the connection and transaction with the database and provides a way to query and manipulate the data.

Entities: Entities are the objects that represent the data in your application. In Entity Framework Core, entities are typically represented as classes, and each entity typically maps to a table in the database.

DbSet: DbSet is a class in Entity Framework Core that represents a collection of entities of a specific type. It provides functionality for querying, adding, updating, and deleting entities.

Code First: Code First is an approach in Entity Framework Core where you define your entity classes and relationships first, and then Entity Framework Core generates the database schema based on those definitions. It allows you to work primarily with classes and objects and provides a fluent API for configuring the mapping between classes and database tables.

Fluent API: Fluent API is a set of configuration methods provided by Entity Framework Core that allow you to customize the mapping between your entity classes and database tables. It provides a fluent, readable syntax for specifying relationships, defining primary and foreign keys, specifying column names, and more.

Migrations: Migrations in Entity Framework Core are a way to manage changes to the database schema over time. They allow you to evolve your database schema as your application evolves. Migrations enable you to create, update, and revert database schema changes using code-based migration files.

Querying: Entity Framework Core provides a rich set of querying capabilities. You can write LINQ (Language Integrated Query) expressions to query the database and retrieve entities. Entity Framework Core translates these LINQ queries into SQL queries and executes them against the database.

Relationships: Entity Framework Core supports various types of relationships between entities, such as one-to-one, one-to-many, and many-to-many. You can define these relationships using navigation properties in your entity classes or using the Fluent API.

Change Tracking: Entity Framework Core tracks changes made to the entities in the DbContext. It automatically detects changes and generates the appropriate SQL statements to persist those changes to the database.

Lazy Loading and Eager Loading: Entity Framework Core supports lazy loading and eager loading. Lazy loading defers the loading of related entities until they are accessed, while eager loading loads the related entities in a single query upfront.

Database Providers: Entity Framework Core supports multiple database providers, including Microsoft SQL Server, SQLite, MySQL, PostgreSQL, and more. Each provider has its own specific configuration and behavior, allowing you to work with different databases seamlessly.

Inheritance Mapping: Entity Framework Core provides support for mapping inheritance hierarchies to database tables. You can use Table-Per-Hierarchy (TPH), Table-Per-Type (TPT), or Table-Per-Concrete-Class (TPC) strategies to define how the inheritance is mapped to the database schema.

Transactions: Entity Framework Core supports transactions, allowing you to perform multiple database operations as a single atomic unit. You can use transactions to ensure data consistency and integrity.

Concurrency Control: Entity Framework Core provides mechanisms for handling concurrency control, allowing multiple users to access and modify the same data concurrently. It supports optimistic concurrency control, where changes are tracked and conflicts are resolved during the save operation.

Database Seeding: Entity Framework Core supports database seeding, which allows you to populate the database with initial data during application startup. You can define seed data using migrations or by overriding the OnModelCreating method in your DbContext.

Query Filters: Query filters in Entity Framework Core enable you to define global filters that are automatically applied to queries. These filters can be used to enforce data security, soft delete functionality, or any other custom filtering requirements.

Value Conversions: Entity Framework Core supports value conversions, allowing you to map non-scalar properties to different representations in the database. For example, you can store an enum as an integer or a string in the database using value conversions.

Owned Entities: Entity Framework Core introduces the concept of owned entities, which are entity types that are exclusively owned by another entity. Owned entities share the same table and lifecycle as their owning entity.

Interceptors: Entity Framework Core provides interceptors that allow you to intercept and modify the behavior of database operations. You can use interceptors to log queries, modify SQL statements, or perform custom actions before or after executing database operations.

Testing: Entity Framework Core offers various strategies for testing applications that use the framework. It provides in-memory databases and the ability to mock DbContext and DbSets, allowing you to write unit tests for your data access layer.

Reverse Engineering: database as source of truth

### Entity

Refers to a class that represents a data model or an object in your application. For example, a "Customer" entity class might have properties like "Id", "Name", "Email", "Address", etc., representing the various attributes of a customer. Acts as a bridge between the object-oriented world of your application and the relational world of the database. The entity classes define the structure and behavior of the data objects that will be persisted in the database. Usually Class property with “Id” string is considered as primary key. [Key] attribute can be used to override a paritcular property as primary key.

[Column(TypeName=”decimal(6,2)”)] to define decimal points with two decimal points.

Each property in the entity class typically corresponds to a column in the table. The framework handles the mapping between the entity objects and the database, allowing you to work with entities in your application's code while transparently persisting the data to the database.

LINQ queries and methods provided by the framework to retrieve, insert, update, and delete entities in a database-agnostic way. The DbContext class acts as a gateway for working with entities, providing methods to query, track changes, and perform database operations on the entities.

### DbContext

The DbContext class in Entity Framework Core provides a wide range of methods for interacting with the database and performing various operations. Here are some important methods in DbContext:

1. Set<TEntity>: This method returns a DbSet<TEntity> instance that represents a collection of entities of a specific type. It allows you to query, insert, update, and delete entities of that type.

2. Find<TEntity>(params object[] keyValues): This method is used to retrieve an entity by its primary key values. It simplifies the process of fetching an entity based on its key values.

3. Add<TEntity>(TEntity entity): This method is used to add a new entity to the context. The entity is initially marked as Added, and the changes are saved to the database when SaveChanges is called.

4. Update<TEntity>(TEntity entity): This method is used to mark an existing entity as modified in the context. It can be used to update the state of an entity before saving changes to the database.

5. Remove<TEntity>(TEntity entity): This method is used to mark an entity as deleted in the context. The entity is scheduled for deletion, and the changes are saved to the database when SaveChanges is called.

6. Attach<TEntity>(TEntity entity): This method is used to attach an entity to the context in an Unchanged state. It is typically used when you have an entity that was created outside the context and you want to start tracking it.

7. SaveChanges: This method is used to persist the changes made to the entities in the context to the underlying database. It executes the appropriate SQL statements for insert, update, and delete operations.

8. Entry<TEntity>(TEntity entity): This method is used to get an EntityEntry<TEntity> object for a given entity. The EntityEntry provides access to various properties and methods for working with the entity's state and tracking information.

9. Query<TEntity>: This method is used to create a query for retrieving entities of a specific type. It returns an IQueryable<TEntity> that can be further refined using LINQ queries.

10. FromSqlRaw/FromSqlInterpolated: These methods are used to execute raw SQL queries against the database. FromSqlRaw accepts a raw SQL query string, while FromSqlInterpolated allows you to use interpolated strings to compose the query.

These are just some of the important methods provided by the DbContext class in Entity Framework Core. The DbContext class offers many more methods and properties for working with entities, relationships, database operations, and configuration.

### Code first approach vs Database first approach

The code-first approach and the database-first approach are two different approaches to developing applications and managing their underlying databases. Let's take a closer look at each approach:

**Code-First Approach:**

In the code-first approach, the primary focus is on developing the application's code, including the models, business logic, and application functionality. The database schema is generated automatically based on the code structure. The developer defines the models and relationships between them using a programming language or an object-relational mapping (ORM) framework.

**Advantages of the code-first approach include:**

Rapid development: Developers can focus on writing code without worrying about the underlying database structure.

Version control: The database schema can be version controlled along with the application code.

Flexibility: Changes to the database schema can be easily made by modifying the code, and migrations can be automatically generated to update the database accordingly.

**Disadvantages of the code-first approach include:**

Potential for inconsistencies: If the database schema is generated automatically based on the code, there is a possibility of inconsistencies or conflicts between the code and the database.

Lack of control over database design: The database schema is heavily influenced by the code structure, which may not always align with optimal database design principles.

Limited support for legacy databases: Code-first approaches may not be ideal when working with existing databases or legacy systems.

Popular frameworks that support code-first development include Entity Framework (for .NET), Django (for Python), and Sequelize (for Node.js).

**Database-First Approach:**

In the database-first approach, the database schema is designed and implemented first, and then the application code is generated or written to work with that existing schema. The developer defines the database structure using tools like database management systems (DBMS) or entity-relationship diagrams (ERDs), and then generates the code or writes the code manually to interact with the database.

**Advantages of the database-first approach include:**

Control over database design: Developers have full control over the database schema design, allowing them to optimize performance, enforce constraints, and ensure data integrity.

Seamless integration with legacy systems: When working with existing databases or legacy systems, the database-first approach allows developers to leverage the existing schema and build the application code around it.

Clear separation of concerns: The database schema can be designed independently of the application code, facilitating better separation of concerns between the database team and the application development team.

Disadvantages of the database-first approach include:

Potential for manual errors: Writing code to match an existing database schema introduces the possibility of manual errors, especially when dealing with complex or large-scale databases.

Increased development time: The need to design and implement the database schema before writing application code may lead to longer development timelines.

Limited flexibility for database changes: Modifying the database schema after the code has been written may require significant effort to update the application code accordingly.

Tools like SQL Server Management Studio (for Microsoft SQL Server), MySQL Workbench (for MySQL), and Oracle SQL Developer (for Oracle) support the database-first approach by providing features to design and generate code from database schemas.

### DbSet

## Sample code

### First Sample app MSDN reference

[Link](https://learn.microsoft.com/en-us/ef/core/get-started/overview/first-app?tabs=netcore-cli)

### Code to create a table for a c# model

To create a table in the database using a C# data model in Entity Framework Core, you need to follow these steps:

1. Define the Data Model: Create a C# class that represents the data model for the table. Each property in the class typically corresponds to a column in the table. For example, let's create a simple "Product" data model with an Id, Name, and Price:

csharp

public class Product

{

public int Id { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }

}

2. Create a DbContext Class: Create a class that derives from DbContext, which acts as a bridge between your application and the database. In this class, you will define a DbSet for the data model you want to create a table for. For example:

csharp

public class YourDbContext : DbContext

{

public DbSet<Product> Products { get; set; }

}

3. Configure Database Connection: In your application's configuration (e.g., appsettings.json or Startup.cs), specify the connection string to the database you want to use.

4. Register DbContext: In your application's startup code (e.g., Startup.cs), register your DbContext with the dependency injection container. For example:

csharp

services.AddDbContext<YourDbContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("YourConnectionString")));

5. Run Database Migrations: Entity Framework Core uses migrations to create and update the database schema based on your data model. Run the following command in the Package Manager Console or the .NET CLI to create the initial migration:

dotnet ef migrations add InitialCreate

This command generates a migration file that contains the instructions for creating the table in the database.

6. Apply Migrations: Run the following command in the Package Manager Console or the .NET CLI to apply the migration and create the table in the database:

dotnet ef database update

This command executes the migration and applies the changes to the database.

After completing these steps, Entity Framework Core will create the table in the database based on your data model. You can then use the DbContext and DbSet to query, insert, update, and delete records from the table using object-oriented programming paradigms.

### Perform migration using nuget package

Performing migrations programmatically in code allows you to automate the migration process and integrate it into your application's logic. In Entity Framework Core, you can perform migrations in code using the `Migrator` class. Here's how you can do it:

1. Reference the Required Packages: Make sure you have the necessary NuGet packages installed for Entity Framework Core. These include `Microsoft.EntityFrameworkCore` and the database provider package, such as `Microsoft.EntityFrameworkCore.SqlServer` for SQL Server.

2. Create an Instance of Your DbContext: Instantiate your DbContext class, which represents the database context for your application. This class should derive from `DbContext` and contain your data models and DbSet properties.

csharp

using Microsoft.EntityFrameworkCore;

public class YourDbContext : DbContext

{

public DbSet<Product> Products { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer("YourConnectionString");

}

}

Replace `"YourConnectionString"` with the appropriate connection string for your database provider.

3. Create the Migration: Use the `Migrator` class to create a migration. You can specify a name for the migration, which helps identify it in the future. Here's an example:

csharp

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Migrations;

public static void CreateMigration()

{

using (var dbContext = new YourDbContext())

{

var migrator = dbContext.GetService<IMigrator>();

migrator.Migrate("YourMigrationName");

}

}

Replace `"YourMigrationName"` with a meaningful name for your migration.

4. Apply the Migration: Use the `Migrator` class to apply the migration to the database. This step will execute the necessary SQL scripts to create or update the database schema.

csharp

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Migrations;

public static void ApplyMigration()

{

using (var dbContext = new YourDbContext())

{

var migrator = dbContext.GetService<IMigrator>();

migrator.Migrate();

}

}

This code will apply the latest migration to the database.

By calling the `CreateMigration` and `ApplyMigration` methods in your application, you can programmatically create and apply migrations without relying on the command prompt. It's important to ensure that your application has the necessary permissions to modify the database schema.