Understanding ORM and Entity Framework Core

- **What is ORM?**

An Object-Relational Mapper (ORM) simplifies database interactions by allowing developers to work with database objects as if they were in-memory objects. It eliminates the need to write complex SQL queries by providing a high-level API.

- **Entity Framework Core:**

  Entity Framework Core (EF Core) is an Object-Relational Mapping (ORM) framework for .NET applications. It allows developers to interact with a database using .NET objects, eliminating the need for most raw SQL queries. With EF Core, you define your database schema using C# classes, and EF Core translates this into SQL commands to perform CRUD operations.

  - Supports both code-first and database-first approaches.

- **How Entity Framework Core Works:**

  - Maps C# classes to database tables.
  - Generates SQL queries to interact with the database.
  - Tracks changes to objects and updates the database when SaveChanges is called.
  - Provides extensibility through Fluent API for fine-tuned control over mappings and configurations.

When using Entity Framework Core (EF Core), there are several processes happening behind the scenes to translate your C# code into database interactions. Here's a step-by-step explanation:

1. Setting Up the `DbContext`

- **Configuration**: When you instantiate a `DbContext` (e.g., `LibraryContext`), EF Core reads the configuration defined in the `OnConfiguring` method or `AddDbContext` in `Startup.cs` (for ASP.NET Core apps).
- **Connection**: It establishes a connection to the database using the connection string provided.

2. Model Building

- **Entity Mapping**: EF Core maps your C# classes (like `Book`) to database tables.
  - **Class-to-Table Mapping**: Each class becomes a table in the database.

- **Property-to-Column Mapping**: Each property becomes a column in the table.
- **Relationships**: Relationships between classes (like one-to-many or many-to-many) are mapped using conventions or explicit configurations (e.g., `HasOne`, `HasMany`).

3. Migrations

When you add a migration, EF Core:

- Compares the current state of your model (C# classes) with the previous migration or the database schema.
- Generates a migration file with SQL commands to synchronize the database schema with your model.

When you apply the migration:

- EF Core executes the SQL commands from the migration file to create or update database objects (tables, columns, keys, etc.).

4. Query Execution

When you perform a query, EF Core:

1. **Expression Parsing**:
   - It takes your LINQ query (e.g., `context.Books.ToList()`) and translates it into an **Expression Tree**.
   - This tree represents the structure and intent of your query.
2. **SQL Generation**:
   - EF Core analyzes the expression tree and translates it into a SQL query.
   - For example:

     ```
     var books = context.Books.Where(b => b.Pages > 200).ToList();
     ```

     Generates:

     ```
     SELECT * FROM Books WHERE Pages > 200;
     ```
3. **Database Interaction**:

- EF Core sends the SQL query to the database using the underlying database provider (e.g., SQL Server, PostgreSQL).
- The database executes the query and returns the results.

4. **Object Materialization**:
   - EF Core takes the results from the database and maps them back to your C# objects (entities).
   - It uses reflection to populate the properties of the objects.

5. Change Tracking

When you retrieve entities from the database:

- EF Core starts tracking changes to these objects in the **Change Tracker**.
- For example:

```
var book = context.Books.FirstOrDefault(b => b.Id == 1);
book.Title = "New Title";
context.SaveChanges();
```

  - EF Core detects that the `Title` property of the `book` object was modified.
  - It generates an `UPDATE` SQL command to persist the changes:

```
UPDATE Books SET Title = 'New Title' WHERE Id = 1;
```

6. Lazy vs. Eager Loading

When dealing with related data:

- **Lazy Loading**: EF Core defers loading related data until it's explicitly accessed. It executes additional queries as needed.
- **Eager Loading**: EF Core includes related data in the initial query using `.Include()`:

```
var books = context.Books.Include(b => b.Author).ToList();
```

  This generates a `JOIN` query to fetch both `Books` and related `Authors`.

7. Transaction Management

- EF Core automatically wraps `SaveChanges()` in a database transaction.
- If something fails (e.g., a constraint violation), it rolls back all changes to maintain data integrity.

EF Core acts as a translator and intermediary between your C# application and the database, simplifying complex operations into straightforward method calls.