1. Introduction to Entity Framework Core (EF Core)

Basics of EF Core

- **Question:** What is Entity Framework Core, and how does it differ from Entity Framework 6?
  Answer: EF Core is a lightweight, cross-platform, open-source version of Entity Framework. It supports more modern features and performance improvements compared to EF 6. EF Core is also modular, supports LINQ queries, and is designed for use in .NET Core and .NET applications, whereas EF 6 only runs on the .NET Framework.

Supported Databases

- **Question:** What databases are supported by Entity Framework Core?
  Answer: EF Core supports SQL Server, SQLite, PostgreSQL, MySQL, Cosmos DB, and other relational databases through third-party providers. It also supports NoSQL databases like Azure Cosmos DB.

---

2. Code-First Approach in EF Core

Code-First Migrations

- **Question:** How do you enable migrations in Entity Framework Core Code-First?
  Answer: Migrations in Code-First can be enabled using the `Add-Migration` and `Update-Database` commands in the package manager console. Migrations track changes to your model and apply them to the database schema.

```bash
Add-Migration InitialCreate Update-Database
```

Fluent API vs Data Annotations

- **Question:** What is the difference between Fluent API and Data Annotations in EF Core?
  Answer: Data Annotations are attributes applied to entity properties and classes to define how they map to the database. Fluent API provides more flexibility and can define complex configurations that Data Annotations cannot, such as composite keys or many-to-many relationships, and is used in the `OnModelCreating` method in the `DbContext`.

---

3. DbContext and DbSet in EF Core

DbContext

- **Question:** What is `DbContext`, and how does it work in EF Core?
  Answer: `DbContext` is the primary class in EF Core that interacts with the database. It represents a session with the database and is used for querying and saving data. `DbContext` manages entities and tracks changes to them for persistence in the database.

DbSet

- **Question:** What is `DbSet` in Entity Framework Core?
  Answer: `DbSet` represents a collection of entities in EF Core, mapping to a table in the database. You use it to query, add, update, and delete entities from the database.

```csharp
```

```
public DbSet<Customer> Customers { get; set; }
```

---

4. LINQ Queries in EF Core

Querying with LINQ

- **Question:** How do you write LINQ queries in EF Core?
  Answer: EF Core allows you to write queries using LINQ (Language Integrated Query). These LINQ expressions are translated into SQL at runtime and executed against the database.

```csharp
var customers = context.Customers.Where(c => c.Age > 30).ToList();
```

Query Syntax vs Method Syntax

- **Question:** What is the difference between query syntax and method syntax in LINQ?
  Answer: Query syntax is similar to SQL-like syntax (e.g., `from c in context.Customers select c`). Method syntax uses method calls like `Where()`, `Select()`, and `OrderBy()` for querying data (e.g., `context.Customers.Where(c => c.Age > 30)`).

---

5. Loading Related Data

Eager Loading

- **Question:** What is eager loading in EF Core, and how is it done?
  Answer: Eager loading loads related data as part of the initial query using the `Include()` method, reducing the number of separate database queries but increasing the size of the query.

```csharp
var orders = context.Orders.Include(o => o.Customer).ToList();
```

Lazy Loading

- **Question:** How does lazy loading work in EF Core?
  Answer: Lazy loading defers the loading of related data until it is accessed. This requires navigation properties to be virtual and EF Core to have lazy loading proxies enabled. When a related entity is accessed, a separate query is made to fetch the data.

Explicit Loading

- **Question:** How is explicit loading implemented in EF Core?
  Answer: Explicit loading is used when you want to manually load related data after the initial query, using methods like `Load()` or `Collection.Load()`.

```csharp
context.Entry(order).Reference(o => o.Customer).Load();
```

---

6. Entity Relationships in EF Core

One-to-Many Relationship

- **Question:** How do you configure a one-to-many relationship in EF Core?
  Answer: A one-to-many relationship can be configured using navigation properties and the `HasMany()` and `WithOne()` methods in the Fluent API or using Data Annotations like `[ForeignKey]`.

```csharp
modelBuilder.Entity<Customer>() .HasMany(c => c.Orders) .WithOne(o => o.Customer);
```

Many-to-Many Relationship

- **Question:** How do you configure a many-to-many relationship in EF Core 5.0 and later?
  Answer: Starting with EF Core 5.0, you can configure many-to-many relationships without an explicit join table by using `HasMany()` and `WithMany()`. EF Core creates a join table automatically.

```csharp
modelBuilder.Entity<Student>() .HasMany(s => s.Courses) .WithMany(c => c.Students);
```

---

7. Concurrency Control

Optimistic Concurrency

- **Question:** How is optimistic concurrency handled in EF Core?
  Answer: EF Core uses concurrency tokens like `RowVersion` to detect when multiple users attempt to update the same record at the same time. If a conflict is detected, EF Core throws a `DbUpdateConcurrencyException`, which can be handled in code.

```csharp
[Timestamp] public byte[] RowVersion { get; set; }
```

Handling Concurrency Conflicts

- **Question:** How do you handle concurrency conflicts in EF Core?
  Answer: You can handle concurrency conflicts by catching `DbUpdateConcurrencyException` and either retrying the operation, merging changes, or refreshing the data from the database.

```csharp
try { context.SaveChanges(); } catch (DbUpdateConcurrencyException) { // Handle concurrency conflict }
```

---

8. Transactions in EF Core

Managing Transactions

- **Question:** How do you manage transactions in EF Core?
  Answer: EF Core automatically manages transactions when `SaveChanges()` is called. However, you can manually manage transactions using `BeginTransaction()`, `Commit()`, and `Rollback()` to ensure atomicity in complex operations.

```csharp
```

```
using var transaction = context.Database.BeginTransaction(); try {
context.SaveChanges(); transaction.Commit(); } catch { transaction.Rollback(); }
```

SaveChanges Behavior

- **Question:** How does `SaveChanges()` work in EF Core with transactions?
  Answer: When `SaveChanges()` is called, EF Core wraps the changes in a transaction and executes them as a single atomic unit. If any part of the transaction fails, the entire transaction is rolled back.

---

9. Migrations in EF Core

Applying Migrations

- **Question:** How do you apply migrations in EF Core?
  Answer: Migrations track model changes and apply them to the database schema. They are applied using the `Add-Migration` and `Update-Database` commands. `Add-Migration` generates migration files, and `Update-Database` applies them to the database.

```bash
Add-Migration AddNewColumn Update-Database
```

Rolling Back a Migration

- **Question:** How do you roll back a migration in EF Core?
  Answer: You can roll back a migration by specifying a target migration using the `Update-Database` command or by reverting all migrations by specifying `0`.

```bash
Update-Database -Migration InitialCreate
```

---

10. Performance Optimization in EF Core

Using `AsNoTracking`

- **Question:** What is `AsNoTracking()` in EF Core, and when should it be used?
  Answer: `AsNoTracking()` is used for read-only queries to disable change tracking, which reduces the overhead of managing entity state. This improves performance when you are only retrieving data and not updating it.

```csharp
var customers = context.Customers.AsNoTracking().ToList();
```

Batch Operations

- **Question:** How do batch operations improve performance in EF Core?
  Answer: EF Core 6.0 introduced batch updates and deletes, allowing multiple rows to be updated or deleted with a single SQL command instead of fetching the entities first. This reduces the number of database round-trips.

```csharp
```

```
context.Customers.Where(c => c.IsInactive).BatchDelete();
```

---

11. Shadow Properties and Global Query Filters

Shadow Properties

Shadow Properties

- **Question:** What are shadow properties in EF Core, and how do you configure them?
  Answer: Shadow properties are properties that exist in the database but are not defined in the entity class. They are typically used for metadata like timestamps. Shadow properties are configured using the Fluent API in the `OnModelCreating` method.

```
modelBuilder.Entity<Customer>().Property<DateTime>("CreatedOn");
```

Accessing Shadow Properties

- **Question:** How do you access shadow properties in EF Core?
  Answer: Shadow properties can be accessed by using `Entry()` and specifying the property name as a string. This is useful for both reading and updating shadow properties.

```
var createdOn = context.Entry(customer).Property("CreatedOn").CurrentValue;
```

---

12. **Global Query Filters**

Defining Global Query Filters

- **Question:** What are global query filters in EF Core, and how are they implemented?
  Answer: Global query filters allow you to apply a filter to all queries for a particular entity type, ensuring that only certain entities are retrieved. This is useful for implementing soft deletes or multi-tenancy. Global query filters are defined in `OnModelCreating`.

```
```

```
modelBuilder.Entity<Customer>().HasQueryFilter(c => !c.IsDeleted);
```

Overriding Global Query Filters

- **Question:** How do you override or disable a global query filter in EF Core?
  Answer: You can override a global query filter by using the `IgnoreQueryFilters()` method in a LINQ query. This ensures that the filter is bypassed when retrieving entities.

```
var allCustomers = context.Customers.IgnoreQueryFilters().ToList();
```

---

13. **Entity Lifecycle Events and Change Tracking**

Tracking Entity Changes

- **Question:** How does change tracking work in EF Core, and what is the role of the `ChangeTracker`?
  Answer: EF Core automatically tracks changes to entity objects by default. The `ChangeTracker` keeps track of the state (Added, Modified, Deleted, Unchanged) of each entity. When `SaveChanges()` is called, EF Core generates the appropriate SQL commands based on these tracked changes.

Detecting Changes Manually

- **Question:** How do you manually detect changes in EF Core?
  Answer: You can use the `ChangeTracker.DetectChanges()` method to manually detect changes to entities if automatic detection is disabled or if you need more control over when changes are detected.

```
context.ChangeTracker.DetectChanges();
```

---

14. **Keyless Entities and Query Types**

Keyless Entities

- **Question:** What are keyless entities in EF Core, and when would you use them?
  Answer: Keyless entities (formerly known as query types) are used to map database queries or views that don't have a primary key. They are typically used for read-only data or complex queries that don't fit into the standard entity model.

```
public class ProductReport { public string ProductName { get; set; } public int
QuantitySold { get; set; } } modelBuilder.Entity<ProductReport>().HasNoKey();
```

Using Keyless Entities in Queries

- **Question:** How do you use keyless entities in a query in EF Core?
  Answer: You can map keyless entities to SQL views or complex queries using `FromSqlRaw()` or `FromSqlInterpolated()`. These queries are often used for reporting purposes.

```
var report = context.Set<ProductReport>().FromSqlRaw("SELECT * FROM
ProductReportView").ToList();
```

---

15. **Performance Tuning in EF Core**

No-Tracking Queries

- **Question:** What is the benefit of using no-tracking queries in EF Core, and when should you use them?
  Answer: No-tracking queries (`AsNoTracking()`) improve performance for read-only data by disabling change tracking. This reduces memory usage and the overhead associated with tracking entity state. It should be used when you don't need to update the queried entities.

  ```
  var customers = context.Customers.AsNoTracking().ToList();
  ```

Compiled Queries

- **Question:** What are compiled queries in EF Core, and how do they improve performance?
  Answer: Compiled queries are precompiled versions of LINQ queries that improve performance by avoiding the cost of compiling the query expression tree on each execution. Compiled queries are useful for frequently executed queries.

  ```
  var query = EF.CompileQuery((MyContext ctx, int id) => ctx.Customers.Where(c =>
  c.Id == id).FirstOrDefault()); var customer = query(context, 1);
  ```

---

16. **Migration Strategies**

Applying Migrations

- **Question:** How do you apply migrations in EF Core?
  Answer: Migrations can be applied using the `Add-Migration` and `Update-Database` commands in the Package Manager Console or the CLI. This generates migration files and updates the database schema to match the current model.

  ```
  Add-Migration InitialCreate Update-Database
  ```

Customizing Migrations

- **Question:** How do you customize migrations in EF Core?
  Answer: After generating a migration, you can customize the migration by editing the `Up()` and `Down()` methods in the generated migration file. This allows you to add custom SQL or logic for schema changes.

---

17. **Handling Exceptions in EF Core**

Common EF Core Exceptions

- **Question:** What are some common exceptions in Entity Framework Core, and how do you handle them?
  Answer: Common exceptions in EF Core include:
  - **DbUpdateException**: Thrown when an error occurs while updating the database.
  - **DbUpdateConcurrencyException**: Thrown when a concurrency conflict occurs.
  - **DbEntityValidationException**: Thrown when entity validation fails. These exceptions can be caught using try-catch blocks, and specific handling can be implemented based on the exception type.

```
try { context.SaveChanges(); } catch (DbUpdateException ex) { // Handle update error }
```

Handling Concurrency Exceptions

- **Question:** How do you handle concurrency exceptions in EF Core?
  Answer: Concurrency exceptions (`DbUpdateConcurrencyException`) are handled by either reloading the data from the database, retrying the operation, or merging the client's changes with the database's version.

```
try { context.SaveChanges(); } catch (DbUpdateConcurrencyException ex) { // Handle concurrency conflict }
```

---

18. **Working with Stored Procedures and Raw SQL**

Calling Stored Procedures

- **Question:** How do you call stored procedures in Entity Framework Core?
  Answer: Stored procedures can be called using the `FromSqlRaw()` or `ExecuteSqlRaw()` methods in EF Core. You can either map the result to an entity or execute a procedure that doesn't return data.

```
var customers = context.Customers.FromSqlRaw("EXEC GetAllCustomers").ToList();
```

Executing Raw SQL

- **Question:** How do you execute raw SQL in EF Core?
  Answer: You can execute raw SQL queries using the `FromSqlRaw()` method for queries or `ExecuteSqlRaw()` for commands that do not return data.

```
var products = context.Products.FromSqlRaw("SELECT * FROM Products").ToList();
```

---

19. **NoSQL and EF Core**

Working with NoSQL Databases

- **Question:** How does Entity Framework Core support NoSQL databases like Cosmos DB?
  Answer: EF Core has built-in support for Azure Cosmos DB, allowing you to work with NoSQL data using the same LINQ queries and models as relational databases. You configure EF Core to use Cosmos DB in the `OnConfiguring` method or during dependency injection.

```
optionsBuilder.UseCosmos("<account-endpoint>", "<auth-key>", "<database-name>");
```

Mapping Entities to Cosmos DB

- **Question:** How do you map entities to Cosmos DB in EF Core?
  Answer: You map entities to Cosmos DB by specifying the container (table equivalent) and partition key using the Fluent API in the `OnModelCreating` method.

```
modelBuilder.Entity<Customer>().ToContainer("Customers").HasPartitionKey(c =>
c.CustomerId);
```

---

20. **Designing for Soft Deletes**

Implementing Soft Deletes

- **Question:** How do you implement soft deletes in Entity Framework Core?
  Answer: Soft deletes can be implemented by adding an `IsDeleted` property to your entity and applying a global query filter to exclude deleted entities from queries. Instead of physically deleting records, the `IsDeleted` flag is set to true.

```
modelBuilder.Entity<Customer>().HasQueryFilter(c => !c.IsDeleted);
```
