Self-Study

1. What are System Design phases ? What is a class diagram ? and in which phase is it considered in building a dot net application ?

System Design phases generally refer to the stages involved in defining the architecture, components, and interfaces for a software system. These phases typically include:

- Conceptual Design: Understanding requirements and defining high-level components of the system.
- Logical Design: Defining logical relationships, data flow, and interactions between system components.
- Physical Design: Translating the logical design into actual hardware and software specifications for implementation.

The System Development Life Cycle (SDLC) broadly includes phases such as Planning, Analysis, Design, Development (Implementation), Testing, Deployment, and Maintenance.

A class diagram is a type of static structure diagram used in software design to describe the system's classes, their attributes, methods, and the relationships between them. It visually models how software classes relate and interact, acting as a blueprint for the system or subsystem. Class diagrams correspond closely with source code classes and are used to model the high-level or detailed structure of the software.

In building a .NET application, class diagrams are considered primarily in the Design phase of the system development life cycle. During this phase, the software’s architecture and components are planned. Class diagrams help to:

- Understand and clarify system requirements.
- Define system structure and relationships between classes.

- Provide a basis for coding by showing detailed class attributes and methods.

2. What is data seeding and what is it used for? What are the various ways to implement it in dot net ?

In C#, data seeding is the process of populating a database with an initial set of data automatically when the database is created or updated. It is commonly used to insert essential or default data that an application needs to function properly, such as reference data, default users, or configuration settings.

Purpose of Data Seeding

- To provide initial data for development and testing environments.
- To ensure the application has necessary default data for proper startup.
- To simplify deployment by automating the population of essential data.

Ways to Implement Data Seeding in .NET

1. Using Entity Framework Core Fluent API (HasData method)
    - Define initial seed data in the `OnModelCreating` method of the DbContext using `modelBuilder.Entity<T>().HasData()`.
    - This method is static and works well for simple and static seed data.
    - Seed data is applied via migrations at database update time, requiring a specified Id for entities.
2. Custom Seeder Classes
    - Create a custom seeder class with a `Seed()` method that programmatically checks and inserts data.
    - This approach is more flexible and allows for querying the database and conditional logic.

- Seeder classes are typically called at application startup.

3. New Methods in EF Core 9: UseSeeding and UseAsyncSeeding
    - Allow dynamic and more complex seeding logic, including querying the database and calling external services.
    - Can be registered in the DbContext configuration to perform seeding synchronously or asynchronously.
    - These methods do not require specifying static keys, as IDs can be generated by the database.
4. Seeding from External Files
    - Using JSON, CSV, or other formats as seed data templates.
    - Seed data is read from these files and inserted programmatically.
    - Useful for managing large or frequently updated seed data sets.
5. Database Scripts
    - SQL scripts run at database initialization or deployment to insert seed data.
    - Useful for complex data and bulk inserts outside the ORM context.

3. What is data seeding and what is it used for? What are the various ways to implement it in dot net ?

The DbContextOptionsBuilder class in Entity Framework Core is used to configure options for a DbContext, such as database provider, connection strings, logging, and performance settings.

Key Functions of DbContextOptionsBuilder with Parameters and Purpose

1. UseSqlServer(**string connectionString, Action<SqlServerDbContextOptionsBuilder> sqlServerOptionsAction = null**)
    - Parameters:
        - connectionString: The connection string to the SQL Server database.
        - sqlServerOptionsAction: Optional delegate to configure SQL Server-specific options.
    - Purpose: Configures the context to use SQL Server. The optional delegate allows setting provider-specific options including EnableRetryOnFailure, CommandTimeout, and EnableActiveResultSets.
2. EnableActiveResultSets(bool enabled)
    - Configured via the SQL Server options delegate.
    - Purpose: Enables or disables Multiple Active Result Sets (MARS) which allows multiple operations to be executed on a single connection (concurrent queries).
    - Usage example inside SQL Server options:

```
optionsBuilder.UseSqlServer(connectionString, opts =>
opts.EnableRetryOnFailure().EnableActiveResultSets(true));
```

    - This allows EF Core to support multiple data readers on one connection.
3. LogTo(Action<string> logger, LogLevel minimumLevel = LogLevel.Information, DbContextLoggerOptions options = null)
    - Parameters:
        - logger: An action delegate that receives the log string output.

        - minimumLevel: The minimum log level to capture (e.g., Debug, Information, Warning).
        - options: (optional) granular settings for logging behavior.
    - Purpose: Configures EF Core to send its logs to a target, such as the Debug window or console.
    - Example to send EF Core logs to Debug output window at debug level:

```
optionsBuilder.LogTo(s => Debug.WriteLine(s), LogLevel.Debug);
```

    - This helps in debugging by capturing EF Core SQL queries, state changes, and other diagnostic info.

4. UseLoggerFactory(ILoggerFactory loggerFactory)
    - Parameter:
        - loggerFactory: A Microsoft.Extensions.Logging factory to create loggers.
    - Purpose: Use a fully configured logger factory for logging EF Core events with control over sinks and filtering.
5. EnableSensitiveDataLogging()
    - No parameters.
    - Enables logging of sensitive application data such as parameter values in SQL queries (for debugging only, not recommended in production).
6. UseQueryTrackingBehavior(QueryTrackingBehavior behavior)
    - Parameter:
        - behavior: Enum to set tracking behavior (e.g., TrackAll or NoTracking).

- Purpose: Controls how queries track entities for change detection.