.Net Core, ADO.net, Dapper, Entity Framework

Interview Questions and Answers

Sai Reddy 5/2/25 Interview QA

.Net Interview Q & A

.NET Core Web API

1. **What is the difference between .NET Core and .NET Framework?**
   - .NET Core is a cross-platform, open-source framework suitable for modern applications, while .NET Framework is Windows-only.
   - .NET Core provides better performance, supports Docker containers, and is modular via NuGet packages.
   - It is designed for building scalable web apps, microservices, and APIs.
   - .NET Framework supports legacy applications and Windows Forms/WPF which .NET Core (prior to .NET 5) does not fully support.
   - Example: A RESTful API can be developed in .NET Core and hosted on Linux, unlike .NET Framework.
2. **Explain the middleware pipeline in .NET Core.**
   - Middleware in .NET Core is software that's assembled into an application pipeline to handle requests and responses.
   - Each component can perform operations before and after the next component.
   - The order of middleware configuration in Program.cs/Startup.cs matters.
   - Middleware examples: UseRouting, UseAuthentication, UseAuthorization, UseEndpoints.
   - Example:

```
app.UseRouting();
app.UseAuthentication();
app.UseAuthorization();
app.UseEndpoints(endpoints => endpoints.MapControllers());
```

3. **How do you create a RESTful API using .NET Core?**
   - Use the ASP.NET Core Web API template in Visual Studio or CLI (dotnet new webapi).
   - Decorate controllers with [ApiController] and actions with HTTP verbs (e.g., [HttpGet]).
   - Use routing to define endpoint URLs.
   - Add necessary services in Program.cs and implement logic in controllers/services.
   - Example:

```
[ApiController]
[Route("api/[controller]")]
public class ProductsController : ControllerBase
{
   [HttpGet]
   public IActionResult Get() => Ok(_productService.GetAll());
}
```

4. **What are the different return types available in Web API controllers?**
   - IActionResult: Returns various result types (Ok, NotFound, BadRequest).
   - ActionResult: Combines IActionResult with a specific return type.
   - Specific types (e.g., string, int, List) for simple APIs.
   - JsonResult, FileResult, ContentResult, etc. for custom formats.

- Example:

```
public ActionResult<Product> GetById(int id)
{
  var product = _context.Products.Find(id);
  return product == null ? NotFound() : Ok(product);
}
```

5. **How do you handle exceptions globally in .NET Core Web API?**
   - Use middleware like UseExceptionHandler or create custom exception middleware.
   - Can also use filters like ExceptionFilterAttribute for controller-level handling.
   - Ensures all unhandled exceptions return meaningful messages and logs.
   - Prevents exposing stack traces in production.
   - Example:

```
app.UseExceptionHandler(errorApp =>
{
  errorApp.Run(async context =>
  {
    context.Response.StatusCode = 500;
    await context.Response.WriteAsync("An error occurred.");
  });
});
```

6. **What is dependency injection, and how is it implemented in .NET Core?**
   - Dependency Injection (DI) is a design pattern for achieving Inversion of Control (IoC).
   - In .NET Core, services are registered in the DI container using AddScoped, AddSingleton, or AddTransient.
   - Services are injected into constructors of controllers or other services.
   - Encourages loose coupling and easier unit testing.
   - Example:

```
services.AddScoped<IProductService, ProductService>();

public ProductsController(IProductService service)
{
  _service = service;
}
```

7. **Explain attribute routing vs conventional routing in Web API.**
   - Conventional routing defines routes centrally (e.g., in Startup.cs or Program.cs).
   - Attribute routing uses annotations directly on controllers/actions.
   - Attribute routing provides better control and readability.
   - Conventional routing uses templates like "{controller}/{action}/{id?}".
   - Example (attribute routing):

```
[Route("api/products")]
[HttpGet("{id}")]
public IActionResult Get(int id) => Ok(_service.GetById(id));
```

8. **How do you version APIs in .NET Core?**
   - Use NuGet package Microsoft.AspNetCore.Mvc.Versioning.
   - Configure API versioning in Program.cs.

- Use attributes like [ApiVersion("1.0")] and route templates like "api/v{version:apiVersion}/products".
- Enables backward compatibility and smooth evolution.
- Example:

```
[ApiVersion("1.0")]
[Route("api/v{version:apiVersion}/products")]
public class ProductsV1Controller : ControllerBase { ... }
```

9. **How do you implement pagination in a Web API?**
   - Accept pageNumber and pageSize as query parameters.
   - Use Skip and Take in LINQ queries to fetch records.
   - Return metadata (total count, current page) in the response if needed.
   - Improves performance for large datasets.
   - Example:

```
var pagedData = _context.Products.Skip((page - 1) * size).Take(size);
```

10. **What are Filters in .NET Core Web API (Action, Authorization, Exception)?**
    - Filters are used to run code before or after controller actions.
    - Action filters – logic before/after actions.
    - Authorization filters – enforce access control.
    - Exception filters – handle unhandled exceptions.
    - Can be global, controller, or action level.
    - Example:

```
public class LogActionFilter : IActionFilter
{
   public void OnActionExecuting(ActionExecutingContext context) { ... }
   public void OnActionExecuted(ActionExecutedContext context) { ... }
}
```

11. **What is the use of the [ApiController] attribute?**
    - Introduced in ASP.NET Core 2.1.
    - Provides automatic model validation, binding source inference, and better routing behavior.
    - Eliminates need to manually check ModelState.IsValid.
    - Applies conventions like binding [FromBody], [FromRoute] automatically.
    - Example:

```
[ApiController]
[Route("api/[controller]")]
public class ProductsController : ControllerBase { ... }
```

12. **How do you handle file uploads and downloads in .NET Core Web API?**
    - File uploads: use IFormFile and Request.Form.Files.
    - File downloads: return FileContentResult or PhysicalFileResult.
    - Save file to disk or stream it.
    - Apply file size/extension validations.
    - Example (upload):

```
[HttpPost("upload")]
public async Task<IActionResult> Upload(IFormFile file)
{
   var path = Path.Combine("uploads", file.FileName);
```

```
    using var stream = new FileStream(path, FileMode.Create);
    await file.CopyToAsync(stream);
    return Ok();
}
```

Entity Framework

1. What is Entity Framework Core and how does it differ from EF6?
Entity Framework Core (EF Core) is a lightweight, cross-platform ORM.
It supports .NET Core, offering better performance and asynchronous operations.
EF Core is modular, making it easier to extend and use in various scenarios.
Unlike EF6, EF Core is open-source and supports a wider range of databases.
Example: await context.Users.ToListAsync(); – EF Core supports async methods for DB operations.

2. What are the different approaches to database modeling (Code-First, Database-First)?

- **Code-First**: Define C# classes, and EF generates the database schema.
- **Database-First**: Generate models from an existing database using tools like Scaffold-DbContext.
- **Model-First**: Define models in an XML file and generate both C# classes and database schema.
  Code-First is used when the database doesn't exist yet, while Database-First is better for working with legacy databases.
  Example: Scaffold-DbContext "ConnectionString" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models.

3. What is a DbContext and how does it work?
DbContext is the main class for interacting with the database in EF Core.
It manages entities, queries, transactions, and changes.
DbContext tracks the state of entities and synchronizes changes with the database.
It is configured to work with your database via connection strings and model configurations.
Example: public DbSet<User> Users { get; set; } – Defines a table in the database.

4. How do you handle migrations in EF Core?
Migrations in EF Core allow changes to the database schema without data loss.
Use Add-Migration to create a migration and Update-Database to apply it.
Each migration contains instructions to update the database schema.
Migrations can be versioned and stored in source control.
Example: Add-Migration AddPhoneNumberToUser – Adds a new column to the table.

5. What are navigation properties and how do you use them?
Navigation properties define relationships between entities (1-to-1, 1-to-many, many-to-many).
They help in querying related data, like the Orders for a Customer.
Navigation properties can be loaded eagerly, lazily, or explicitly.
EF Core uses these properties for handling relationships and foreign keys.
Example: public ICollection<Order> Orders { get; set; } – Defines a collection of orders for a user.

6. Explain eager loading, lazy loading, and explicit loading.

- **Eager Loading**: Loads related entities with the main entity in a single query using .Include().
- **Lazy Loading**: Loads related data when the navigation property is accessed (requires proxies).
- **Explicit Loading**: Loads related data after the main entity is loaded using context.Entry().Collection().Load().

Choose eager loading for performance with small data sets, lazy for large, and explicit when you need to load on-demand.
Example: context.Users.Include(u => u.Orders).ToList(); – Eager loading.

7. How do you write LINQ queries in EF Core for complex joins?
EF Core allows you to perform joins using LINQ with .Join() or query syntax.
Use multiple joins for complex queries, and ensure proper grouping or filtering.
It supports inner, left, and cross joins.
You can use navigation properties in LINQ for cleaner syntax.
Example:

```
var query = from u in context.Users
        join o in context.Orders on u.Id equals o.UserId
        select new { u.Name, o.Total };
```

8. How do you manage transactions in EF Core?
EF Core provides DbContext.Database.BeginTransaction() to start a transaction.
Wrap multiple changes into one transaction for atomicity.
Use Commit() to save or Rollback() to undo.
It ensures consistency when performing multiple operations on the database.
Example:

```
using var tx = context.Database.BeginTransaction();
context.SaveChanges();
tx.Commit();
```

9. What are value converters in EF Core?
Value converters enable mapping non-primitive types to database-compatible types.
They are used for enums, custom types, or encryption/decryption.
They are configured in the OnModelCreating method using .HasConversion().
Value converters help store custom types as strings or integers in the DB.
Example:

```
modelBuilder.Entity<User>()
  .Property(u => u.Status)
  .HasConversion(v => v.ToString(), v => Enum.Parse<StatusEnum>(v));
```

10. How do you seed initial data using EF Core?
Initial data can be seeded in OnModelCreating using .HasData().
This is useful for default roles, settings, or other required data.
Seed data is applied during migration updates.
It ensures consistency across environments.
Example:

```
modelBuilder.Entity<User>().HasData(new User { Id = 1, Name = "Admin" });
```

Dapper

1. What is Dapper and how is it different from Entity Framework?

Dapper is a lightweight, high-performance Object-Relational Mapper (ORM) for .NET.
It allows for direct SQL queries while mapping query results to objects.
Unlike Entity Framework, Dapper does not manage the database schema and lacks LINQ support.
It is faster than EF for simple queries due to its low overhead.

Example:

```
var users = connection.Query<User>("SELECT * FROM Users").ToList();
```

2. How do you perform basic CRUD operations using Dapper?

Dapper allows CRUD operations through direct SQL queries, with Execute, Query, or QuerySingle methods.
For **Create**: Use Execute to run INSERT statements.
For **Read**: Use Query to retrieve data.
For **Update**: Use Execute to run UPDATE statements.

Example:

```
connection.Execute("UPDATE Users SET Name = @Name WHERE Id = @Id", new { Name = "John", Id = 1 });
```

3. How do you handle stored procedures in Dapper?

Dapper can execute stored procedures using the Execute or Query methods.
You specify the stored procedure name and parameters.
The result can be a scalar value, multiple rows, or nothing.
Stored procedures offer abstraction and better performance for complex queries.

Example:

```
var result = connection.Query<User>("sp_GetUserById", new { Id = 1 }, commandType: CommandType.StoredProcedure);
```

4. What are the pros and cons of using Dapper over EF?

Pros:

- Faster than EF due to minimal overhead.
- More control over SQL queries and database interaction.
- Simpler for small applications or microservices.

Cons:

- Lacks features like change tracking and migrations.
- Requires manual handling of relationships and more code for complex queries.

Example:

```
var user = connection.QueryFirstOrDefault<User>("SELECT * FROM Users WHERE Id = @Id", new { Id = 1 });
```

5. How do you map one-to-many or many-to-many relationships in Dapper?

Dapper can map one-to-many relationships by using Query with multiple result sets.
Use Query to load parent data and related data using SQL joins.
For many-to-many, join tables and map results to a list.
You need to manually manage the relationships.

Example:

```
var userOrders = connection.Query<User, Order, User>(
```

```
"SELECT * FROM Users u JOIN Orders o ON u.Id = o.UserId",
(user, order) => { user.Orders.Add(order); return user; },
splitOn: "UserId").ToList();
```

6. How do you handle parameterized queries in Dapper to prevent SQL injection?

Dapper uses parameterized queries, preventing SQL injection by separating SQL code from data.
You pass parameters as anonymous objects or a dictionary.
This ensures user input is treated as data, not executable code.
Always use parameters in queries to enhance security.

Example:

```
var users = connection.Query<User>("SELECT * FROM Users WHERE Name = @Name",
new { Name = "John" }).ToList();
```

ADO.Net

1. What is ADO.NET and what are its core components?

ADO.NET is a data access framework in .NET for interacting with relational databases.
Its core components include Connection, Command, DataReader, DataSet, and DataAdapter.
Connection manages the database connection, while Command executes SQL queries.
DataReader provides fast, forward-only data retrieval, and DataSet holds data in memory.

Example:

```
using (var connection = new SqlConnection(connectionString))
{
   connection.Open();
   var command = new SqlCommand("SELECT * FROM Users", connection);
   var reader = command.ExecuteReader();
}
```

2. What is the difference between DataSet and DataReader?

- **DataSet**: An in-memory representation of a complete database with support for multiple tables and relationships.
  It can be disconnected and used offline.
- **DataReader**: A forward-only, read-only stream of data from the database, which is faster but requires an active connection.
  DataReader is used for fast, one-time access to data, while DataSet is used for more complex, disconnected operations.
  Example:

```
using (var reader = command.ExecuteReader()) {
   while (reader.Read()) { var name = reader["Name"]; }
}
```

3. How do you execute a stored procedure using ADO.NET?

To execute a stored procedure, you use SqlCommand and set the CommandType to StoredProcedure.
You can pass parameters to the procedure using Parameters.Add.
After execution, you can retrieve results with ExecuteReader, ExecuteNonQuery, or ExecuteScalar.

This allows for interaction with pre-defined database logic.

Example:

```
var command = new SqlCommand("sp_GetUserById", connection)
{
   CommandType = CommandType.StoredProcedure
};
command.Parameters.Add(new SqlParameter("@UserId", 1));
connection.Open();
var result = command.ExecuteScalar();
```

4. How do you manage connection pooling in ADO.NET?

Connection pooling is enabled by default in ADO.NET.

It reuses existing database connections to improve performance.

You can control pooling settings like Max Pool Size and Min Pool Size via the connection string.

Pool size should be configured based on your application's expected load.

Example:

```
string connectionString = "Server=myServerAddress;Database=myDataBase;User
Id=myUsername;Password=myPassword;Max Pool Size=100;";
```

5. How do you handle transactions manually in ADO.NET?

Transactions in ADO.NET are managed using the SqlTransaction object.

You begin a transaction with connection.BeginTransaction(), then commit or roll back using Commit() or Rollback().

This ensures that multiple operations succeed or fail together.

Transactions are useful for maintaining consistency in multi-step processes.

Example:

```
using (var transaction = connection.BeginTransaction())
{
   var command = connection.CreateCommand();
   command.Transaction = transaction;
   command.CommandText = "UPDATE Users SET Name = 'John' WHERE Id = 1";
   command.ExecuteNonQuery();
   transaction.Commit();
}
```

6. What are the potential pitfalls of using ADO.NET in high-load scenarios?

- **Connection Pooling**: Too many open connections can saturate the pool, leading to performance degradation.
- **Concurrency Issues**: ADO.NET doesn't handle concurrency issues like EF Core, which may require manual conflict resolution.
- **Resource Management**: Not properly closing connections can lead to resource leaks.
- **Manual SQL**: Writing raw SQL queries increases the risk of SQL injection and other security concerns.

  Example:

  ```
  // Ensure to close connections to avoid resource leaks
  connection.Close();
  ```

Security Related

1. How do you implement authentication and authorization in .NET Core Web API?

Authentication is typically done using JWT (JSON Web Tokens), which is issued upon login.
The token is validated in the Startup.cs using AddJwtBearer for securing API routes.
Authorization is handled using roles or policies in combination with [Authorize] attributes.
The user must have valid credentials to access restricted resources.

Example:

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
  .AddJwtBearer(options => { options.TokenValidationParameters = new
TokenValidationParameters { /* Validation Settings */ }; });
```

2. What is the difference between JWT and OAuth2?

- **JWT**: A token format used to represent claims securely between two parties.
- **OAuth2**: A protocol for delegation, allowing third-party applications to access resources without exposing credentials.
  JWT is commonly used in OAuth2 as the token format for authorization.
  OAuth2 is broader, handling various authorization flows (client credentials, authorization code, etc.).
  Example:
  OAuth2 grants access tokens, often in the form of JWT, after successful authorization.

3. How do you protect APIs from CSRF, XSS, and SQL injection?

- **CSRF**: Use anti-forgery tokens ([ValidateAntiForgeryToken]) and check the Origin header.
- **XSS**: Sanitize user inputs using HtmlEncode and escape outputs to prevent malicious scripts.
- **SQL Injection**: Use parameterized queries or ORMs like Entity Framework to prevent direct SQL manipulation.
  These methods mitigate attacks and ensure data integrity and security.
  Example:

```
var user = dbContext.Users.Where(u => u.UserName == username).FirstOrDefault();
```

4. How do you secure sensitive data in configuration files (e.g., connection strings)?

Sensitive data, like connection strings, should be stored in environment variables or a secure vault (Azure Key Vault).
Avoid storing such data directly in configuration files or code.
Use the IConfiguration interface to retrieve data from external sources like environment variables.
For additional security, encrypt sensitive settings.

Example:

```
var connectionString =
Configuration.GetValue<string>("ConnectionStrings:DefaultConnection");
```

5. What are CORS and how do you configure it in .NET Core?

CORS (Cross-Origin Resource Sharing) allows or restricts resources to be accessed from a different domain.
In .NET Core, you configure CORS in the Startup.cs file using AddCors.
You can define allowed origins, headers, and methods for different routes.
This ensures your API can be securely accessed by trusted sources.

Example:

```
services.AddCors(options => options.AddPolicy("AllowAll", builder =>
builder.AllowAnyOrigin().AllowAnyMethod().AllowAnyHeader()));
```

6. How do you implement role-based and policy-based authorization in .NET Core?

Role-based Authorization uses roles defined in the ClaimsPrincipal to restrict access.

Policy-based Authorization uses custom policies with more complex conditions.

You configure policies in Startup.cs and protect controllers with [Authorize] and role or policy attributes.

Both methods ensure only authorized users can access specific resources.

Example:

```
services.AddAuthorization(options => {
   options.AddPolicy("AdminOnly", policy => policy.RequireRole("Admin"));
});
[Authorize(Policy = "AdminOnly")]
public IActionResult GetAdminData() { return Ok(); }
```