✅.NET Core Interview Questions (All Levels)

🔰 Basic Level

1. **What is .NET Core? How is it different from .NET Framework?**

Answer:
.NET Core is a **cross-platform**, **open-source**, and **lightweight** framework developed by Microsoft for building modern applications, including web, cloud, and console apps.

Key differences from .NET Framework:

- .NET Core supports **Windows, Linux, and macOS**, whereas .NET Framework is **Windows-only**.
- It provides **high performance and scalability**, especially for web APIs and microservices.
- **Side-by-side versioning** is supported in .NET Core, which is not available in the .NET Framework.
- It's also more modular, using NuGet packages for just what you need.

2. **What are the advantages of using .NET Core?**

Answer:

- **Cross-platform**: Runs on Windows, Linux, and macOS.
- **High performance**: Optimized for modern workloads and microservices.
- **Modular and lightweight**: Uses NuGet-based packages so apps only include what's needed.
- **Built-in Dependency Injection** support.
- **Fast development and deployment** via CLI, Docker, and Azure DevOps integration.
- **Side-by-side versioning**: Multiple versions of .NET Core can run on the same machine.
- **Open-source and active community support**.

3. **What is the use of Program.cs and Startup.cs files in a .NET Core project?**

Answer:

- Program.cs: Entry point of the application. It creates the **host**, configures logging, dependency injection, and web server (Kestrel). Example:

  ```
  csharp
  CopyEdit
  public static void Main(string[] args)
  {
    CreateHostBuilder(args).Build().Run();
  }
  ```

- Startup.cs: Contains application **startup logic**, like:
  - ConfigureServices(): Registers services (DI container).
  - Configure(): Sets up middleware (request pipeline).

Together, they define how the application initializes and handles HTTP requests.

4. **What is Kestrel in .NET Core?**

   Answer:
   Kestrel is a **cross-platform, high-performance web server** built into ASP.NET Core. It handles HTTP requests and serves as the default web server.

   In production, it's often used **behind a reverse proxy** like **IIS or NGINX** for features like SSL termination, load balancing, and header forwarding.

5. **What is a NuGet package and how do you use it?**

Answer:
A NuGet package is a **compiled library (DLL)** bundled with metadata, used to share reusable code across .NET projects. Examples: Newtonsoft.Json, EntityFrameworkCore.

To use:

- Via CLI: dotnet add package <PackageName>
- Or through **Visual Studio** → Manage NuGet Packages → Browse → Install

I often use NuGet for libraries like AutoMapper, Serilog, Swashbuckle (Swagger), etc.

6. **What are Middlewares in .NET Core?**

Answer:
Middlewares are components in the **HTTP request pipeline** that handle requests/responses. They can:

- Log requests
- Authenticate/authorize
- Serve static files
- Handle errors, etc.

Each middleware calls the **next one in the pipeline** or short-circuits it.

Example:

```
csharp
CopyEdit
app.UseAuthentication();
app.UseRouting();
app.UseAuthorization();
```

I've also written **custom middleware** for logging and request validation.

7. **What is Dependency Injection (DI)? How is it implemented in .NET Core?**

   Answer:
   Dependency Injection is a **design pattern** used to manage object dependencies, improving **testability, maintainability, and modularity**.

In .NET Core, it's built-in. You register services in ConfigureServices():

```
csharp
CopyEdit
services.AddScoped<IMyService, MyService>();
```

Then inject it via constructor:

```
csharp
CopyEdit
public MyController(IMyService service)
{
   _service = service;
}
```

I use DI to inject repositories, services, and configuration values.

8. **What are the common return types for Web APIs?**

Answer:

- IActionResult: Generic return type supporting various responses like Ok(), BadRequest(), etc.
- ActionResult<T>: Combines IActionResult and a specific data type for better type safety.
- Task<T> or Task<IActionResult>: Used for asynchronous methods.

Example:

```
csharp
CopyEdit
public async Task<ActionResult<User>> GetUser(int id)
{
   var user = await _userService.GetById(id);
   return Ok(user);
}
```

9. **How do you handle exceptions in .NET Core?**

Answer:

- At a **global level**, I use UseExceptionHandler() or UseDeveloperExceptionPage() in Startup.cs.
- For Web APIs, I implement **global exception handling middleware** that logs errors and returns proper HTTP status codes.
- I also use **try-catch blocks** within services for anticipated exceptions.

Example:

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

```
  });
});
```

For production apps, I integrate with **Serilog** or **Application Insights** for centralized logging.

 Intermediate Level

1. **Explain the ASP.NET Core pipeline.**

Answer:
The ASP.NET Core pipeline is a sequence of **middleware components** that process HTTP requests and responses.

- Each middleware can perform operations on the request, pass it to the next middleware, and process the response.
- It's configured in Startup.cs → Configure() method.

Example pipeline:

```
csharp
CopyEdit
app.UseRouting();
app.UseAuthentication();
app.UseAuthorization();
app.UseEndpoints(endpoints =>
{
  endpoints.MapControllers();
});
```

In my projects, I use this pipeline to handle **logging**, **authentication**, **CORS**, **exception handling**, etc.

2. **How does routing work in .NET Core Web API?**

Routing in ASP.NET Core maps **incoming URLs to controller actions** using route templates.

- Defined via **attribute routing** (preferred in Web APIs):

  ```
  csharp
  CopyEdit
  [Route("api/[controller]")]
  public class ProductsController : ControllerBase
  {
    [HttpGet("{id}")]
    public IActionResult Get(int id) { ... }
  }
  ```

- Or using **conventional routing** in Startup.cs with MapControllerRoute.

.NET Core uses UseRouting() and UseEndpoints() to enable routing.

3. **What is Model Binding in .NET Core?**

Answer:
Model binding is the process where ASP.NET Core **automatically maps HTTP request data** (from query string, form data, body, etc.) to method parameters or models.

Example:

csharp
CopyEdit
[HttpPost]
public IActionResult Create([FromBody] Product product)

Here, the framework binds JSON data from the request body to the Product object.

Model binding supports:

- [FromQuery], [FromRoute], [FromBody], [FromForm], etc.

4. **What is the difference between IActionResult and ActionResult<T>?**

   Answer:

Feature	IActionResult	ActionResult<T>
Return Type	Flexible response types	Strongly typed + flexible response
Best Use	When returning various result types	When returning a specific model + status
Benefits	Good for custom control flow	Type safety + Swagger documentation

Example:
csharp
CopyEdit
public IActionResult Get() => Ok();
public ActionResult<Product> Get(int id) => Ok(product);
In my projects, I prefer ActionResult<T> for clarity and better API docs (e.g., Swagger).

5. **How do you implement authentication and authorization in .NET Core?**

Answer:
I've implemented both **JWT-based** and **cookie-based** authentication.

Steps:

1. **Authentication**: Configure JWT/cookie authentication in Startup.cs.
2. **Authorization**: Use [Authorize] attribute or policy-based access.

Example (JWT):

csharp
CopyEdit
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(...);

app.UseAuthentication();
app.UseAuthorization();

- I use claims-based roles and policies for fine-grained control:

```
csharp
CopyEdit
[Authorize(Roles = "Admin")]
```

6. **What are configuration providers in .NET Core?**

Answer:
Configuration providers are sources from which .NET Core reads configuration settings.

Examples:

- appsettings.json
- Environment variables
- Command-line arguments
- Secret Manager (for dev)
- Azure Key Vault (for production)

They're loaded in CreateHostBuilder():

```
csharp
CopyEdit
.ConfigureAppConfiguration((ctx, config) =>
{
   config.AddJsonFile("appsettings.json")
       .AddEnvironmentVariables();
});
```

I've used different providers depending on **deployment environments** and **security needs**.

7. **What is the role of appsettings.json? How do you use it for environment-specific settings?**

Answer:
appsettings.json is the primary **configuration file** for storing settings like connection strings, logging, and custom keys.

For environments:

- Use files like appsettings.Development.json, appsettings.Production.json
- Environment is set using ASPNETCORE_ENVIRONMENT variable.

Example:

```
json
CopyEdit
"ConnectionStrings": {
  "Default": "Server=.;Database=MyDb;Trusted_Connection=True;"
}
```

In Startup.cs:

```
csharp
```

CopyEdit
var conn = Configuration.GetConnectionString("Default");

I've used this for **multi-environment deployments** via Azure DevOps.

8. **How do you create a custom middleware in .NET Core?**

Answer:
A custom middleware handles HTTP requests in a custom way before passing to the next component.

Steps:

1. Create a class with InvokeAsync(HttpContext context, RequestDelegate next)
2. Register it in Configure()

Example:

csharp
CopyEdit
```
public class LoggingMiddleware
{
   private readonly RequestDelegate _next;
   public LoggingMiddleware(RequestDelegate next) => _next = next;

   public async Task InvokeAsync(HttpContext context)
   {
      // Log something
      await _next(context);
   }
}
```

Register:

csharp
CopyEdit
```
app.UseMiddleware<LoggingMiddleware>();
```

I've created middleware for **request logging**, **header validation**, and **global exception handling**.

9. **Explain the difference between transient, scoped, and singleton services.**

Answer:

Lifetime	Description	Use Case Example
Transient	New instance every time it's requested	Lightweight, stateless services
Scoped	Same instance per HTTP request	EF Core DbContext, unit of work
Singleton	One instance for the application's lifetime	Caching, configuration services

Registration:
csharp
CopyEdit
```
services.AddTransient<IMyService, MyService>();
services.AddScoped<IMyRepo, MyRepo>();
services.AddSingleton<ILogger, MyLogger>();
```

I carefully choose lifetimes to avoid issues like **DbContext scope conflicts**.

10. **What is Entity Framework Core? How do you use it in .NET Core?**

Answer:
EF Core is a lightweight ORM that allows interacting with databases using **C# classes instead of SQL**.

Common steps:

1. Create models and DbContext.
2. Configure DbContext in Startup.cs.
3. Use DbContext in services/repositories.
4. Perform CRUD with LINQ.

Example:

csharp
CopyEdit
services.AddDbContext<AppDbContext>(options =>
  options.UseSqlServer(Configuration.GetConnectionString("Default")));

In code:

csharp
CopyEdit
var products = _context.Products.Where(p => p.Price > 100).ToList();

I use **code-first migration**, **LINQ**, and **asynchronous queries** with EF Core in my projects.

 Advanced Level

1. **How do you implement logging in .NET Core using ILogger?**

Answer:
.NET Core provides built-in logging via `ILogger<T>`. It supports various providers like Console, Debug, EventLog, and third-party tools like **Serilog**, **NLog**, or **Application Insights**.

Setup:

```
csharp
CopyEdit
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;
    public HomeController(ILogger<HomeController> logger)
    {
        _logger = logger;
    }

    public IActionResult Index()
    {
        _logger.LogInformation("Index page accessed.");
        return View();
    }
```

```
}
```

I usually configure logs in `appsettings.json` and in production, I use **Serilog** with **rolling file** or **Azure Log Analytics**.

2. **What is the Unit of Work and Repository Pattern? How have you implemented them?**

Answer:
The **Repository Pattern** abstracts data access logic, and **Unit of Work** coordinates multiple repositories using a shared DbContext to handle transactions.

Implementation:

- `IRepository<T>` with methods like `Add`, `Update`, `GetById`, etc.
- `IUnitOfWork` to group multiple repositories and manage commit/rollback.

```
csharp
CopyEdit
public interface IUnitOfWork : IDisposable
{
    IProductRepository Products { get; }
    IOrderRepository Orders { get; }
    int Complete();
}
```

In my project, this helped maintain **separation of concerns**, especially when working with complex transactions across entities.

3. **How do you handle database migrations in EF Core?**

Answer:
I use **code-first migrations** to evolve the database schema alongside the models.

Commands:

```
bash
CopyEdit
dotnet ef migrations add AddProductTable
dotnet ef database update
```

Steps:

1. Define changes in models.
2. Run `Add-Migration`.
3. Run `Update-Database`.

For multiple environments, I maintain **separate connection strings** and use **migration history table** for version control.

4. **Explain asynchronous programming in .NET Core using async and await.**

Answer:
Async programming in .NET Core improves scalability by **freeing up threads during I/O-bound operations** like DB or API calls.

Example:

```
csharp
CopyEdit
public async Task<IActionResult> GetProducts()
{
    var products = await _productService.GetAllAsync();
    return Ok(products);
}
```

- `async` makes the method asynchronous.
- `await` pauses execution until the task completes.

This is critical in web APIs to avoid thread starvation under load.

5. **What is CQRS and have you used it in any projects?**

Answer:
CQRS (Command Query Responsibility Segregation) separates read (queries) and write (commands) logic into different models.

- **Commands**: Change state (create, update, delete).
- **Queries**: Return data only, no side effects.

In my project, I used CQRS with **MediatR**:

- Queries and commands were handled by separate handlers.
- Improved performance, especially for read-heavy modules.

Used along with **Event Sourcing** in one module to decouple services.

6. **How do you implement caching in ASP.NET Core?**

Answer:
I've implemented both **in-memory caching** and **distributed caching**.

In-Memory Caching:

```
csharp
CopyEdit
services.AddMemoryCache();

public class ProductService
{
    private readonly IMemoryCache _cache;
    public ProductService(IMemoryCache cache)
    {
        _cache = cache;
    }

    public Product GetProduct(int id)
    {
        return _cache.GetOrCreate($"product_{id}", entry => {
            entry.AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(5);
            return _repository.GetById(id);
        });
    }
}
```

Also used **Distributed Redis Cache** for load-balanced apps hosted on Azure.

7. **What are filters in ASP.NET Core (Action Filter, Exception Filter)?**

Answer:
Filters allow code to run **before or after** action methods.

- **Action Filters**: Run before/after controller actions (e.g., logging, validation).
- **Exception Filters**: Handle unhandled exceptions globally.
- **Authorization Filters**: Enforce security policies.

Example:

```
csharp
CopyEdit
public class LogActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context) { /* log */ }
    public void OnActionExecuted(ActionExecutedContext context) { /* log */ }
}
```

I've used **global exception filters** to centralize error handling and **custom action filters** for logging and auditing.

8. **How do you secure Web APIs in .NET Core?**

Answer:
I usually implement **JWT token-based authentication** with role-based access control.

Steps:

- Use `Microsoft.AspNetCore.Authentication.JwtBearer`
- Configure authentication in `Startup.cs`
- Apply `[Authorize]` attribute and custom policies

Example:

```
csharp
CopyEdit
[Authorize(Roles = "Admin")]
public IActionResult GetAllUsers() { ... }
```

Additionally:

- Use **HTTPS redirection**
- Protect sensitive data with **Azure Key Vault**
- Enable **CORS policies**
- Validate inputs to prevent **injection attacks**

9. **Have you used SignalR? What is it and what are its use cases?**

Answer:
Yes. **SignalR** is a library for **real-time communication** between client and server using WebSockets (or fallback protocols).

Use cases:

- Live chat apps
- Real-time dashboards/notifications
- Collaborative apps (e.g., whiteboards, docs)

In my project, I used SignalR to push real-time order status updates to the frontend without polling.

Setup:

```
csharp
CopyEdit
services.AddSignalR();
app.UseEndpoints(endpoints => {
    endpoints.MapHub<ChatHub>("/chathub");
});
```

10. **Explain how CI/CD works with .NET Core applications.**

Answer:
CI/CD automates **build, test, and deployment** processes. I've used **Azure DevOps pipelines** for .NET Core apps.

CI Process:

- Trigger on push/PR
- Restore packages → Build solution → Run unit tests → Generate artifacts

CD Process:

- Pick artifact from CI
- Deploy to Azure Web App or IIS
- Use stages: Dev → QA → Production
- Approvals & rollback configured

YAML Example:

```
yaml
CopyEdit
trigger:
  branches:
    include: [main]

pool:
  vmImage: 'windows-latest'

steps:
- task: UseDotNet@2
  inputs:
    packageType: 'sdk'
    version: '7.x'

- task: DotNetCoreCLI@2
  inputs:
    command: 'build'
    projects: '**/*.csproj'
```

This setup gave us **fast, repeatable, and traceable deployments**.

-