1. Introduction to .NET Framework

Basics of .NET Framework

- **Question:** What is the .NET Framework, and why is it important?
  Answer: The .NET Framework is a software development platform developed by Microsoft that provides a controlled programming environment for building and running Windows applications. It includes a large class library (FCL) and supports languages like C#, VB.NET, and F#. It is important because it offers simplified development for desktop, web, and cloud applications with managed execution, memory management, and security.

CLR (Common Language Runtime)

- **Question:** What is the role of the Common Language Runtime (CLR) in .NET Framework?
  Answer: The CLR is the execution engine of the .NET Framework, responsible for running applications and providing services like memory management (garbage collection), security, exception handling, and thread management.

Assemblies

- **Question:** What are assemblies in .NET Framework?
  Answer: Assemblies are the building blocks of .NET applications. They contain compiled code in the form of IL (Intermediate Language), metadata, and resources like images. Assemblies are either **DLL** (Dynamic Link Libraries) or **EXE** (Executable) files.

---

2. .NET Core Overview

Cross-Platform Capabilities

- **Question:** What is .NET Core, and how is it different from the .NET Framework?
  Answer: .NET Core is a cross-platform, open-source framework for building applications that run on Windows, Linux, and macOS. It differs from the .NET Framework in that it is modular, lightweight, and designed to support cloud-based, microservices, and containerized applications. .NET Core uses NuGet packages for delivering libraries and tools, whereas the .NET Framework has a larger monolithic design.

Command-Line Interface (CLI)

- **Question:** What is the .NET Core CLI, and how is it used?
  Answer: The .NET Core CLI is a cross-platform command-line interface used to develop, build, run, and publish .NET Core applications. It provides commands like `dotnet new` (to create projects), `dotnet build`, `dotnet run`, and `dotnet publish`.

Modular Design

- **Question:** How is the modular design of .NET Core different from .NET Framework?
  Answer: .NET Core uses a modular approach where only the required NuGet packages are included in the application, reducing the size of the application and its dependencies. The .NET Framework includes a large, monolithic base class library, even if only part of it is used by the application.

---

3. .NET (formerly .NET Core)

Unified Platform

- **Question:** What is the new .NET (formerly .NET Core), and what are its key features?
  Answer: The new .NET (starting with .NET 5) is a unified platform that combines .NET Framework, .NET Core, and Xamarin into a single product. It offers cross-platform capabilities, better performance, and a smaller memory footprint. It is designed for all types of applications, including web, desktop, mobile, and cloud-based applications.

.NET Standard

- **Question:** What is .NET Standard, and why is it important?
  Answer: .NET Standard is a formal specification of APIs that all .NET implementations must support. It allows code sharing across different .NET platforms like .NET Core, .NET Framework, Xamarin, and Unity, ensuring a consistent development experience across different environments.

---

4. Garbage Collection and Memory Management

Garbage Collection

- **Question:** How does garbage collection work in the .NET Framework and .NET Core?
  Answer: Garbage collection (GC) in .NET automatically manages memory by reclaiming objects that are no longer referenced by the application. The .NET GC is non-deterministic, meaning it runs at unspecified times, but it is optimized for performance. The GC operates in generations (Gen 0, Gen 1, Gen 2) to manage short-lived and long-lived objects efficiently.

Finalization and IDisposable

- **Question:** What is the difference between finalization and implementing `IDisposable` in .NET?
  Answer: Finalization is used to release unmanaged resources when the object is garbage collected. The `IDisposable` interface allows deterministic disposal of resources by calling `Dispose()` explicitly or using the `using` statement. Finalization is slower because the garbage collector must run twice to clean up the object, while `Dispose()` allows immediate resource cleanup.

---

5. Libraries and Frameworks in .NET

Base Class Library (BCL)

- **Question:** What is the Base Class Library (BCL) in .NET?
  Answer: The BCL is a subset of the .NET Framework Class Library (FCL) that provides fundamental classes like collections, I/O, strings, and types. It serves as the foundation for all .NET applications by offering essential functionality such as data types, file operations, and basic networking.

NuGet Packages

- **Question:** What is NuGet, and how does it relate to .NET Core and .NET?
  Answer: NuGet is a package manager for .NET that allows developers to share and consume reusable components (packages). In .NET Core and .NET, libraries and dependencies are distributed as NuGet packages, making applications modular and lightweight.

---

6. Cross-Platform Development with .NET Core

Cross-Platform Development

- **Question:** How does .NET Core enable cross-platform development?
  Answer: .NET Core allows developers to build and run applications on multiple operating systems like Windows, Linux, and macOS. The cross-platform runtime and libraries in .NET Core ensure that the same application code can be compiled and executed on different platforms without modification.

Docker and .NET Core

- **Question:** How is Docker used with .NET Core applications?
  Answer: Docker is used to containerize .NET Core applications, enabling them to run consistently across different environments. A Dockerfile is used to create a container image for the application, and it can be deployed on any system running Docker. The lightweight nature of .NET Core makes it ideal for microservices and containerized deployments.

---

7. .NET Core and Web Development

ASP.NET Core

- **Question:** What is ASP.NET Core, and how does it differ from ASP.NET in the .NET Framework?
  Answer: ASP.NET Core is a lightweight, cross-platform framework for building modern web applications, APIs, and microservices. Unlike ASP.NET in the .NET Framework, ASP.NET Core is modular, allowing developers to use only the necessary components. It also offers improved performance, better integration with modern cloud and container-based architectures, and simplified configuration.

Middleware in ASP.NET Core

- **Question:** What is middleware in ASP.NET Core, and how does it function?
  Answer: Middleware in ASP.NET Core is a pipeline of components that handle HTTP requests and responses. Each middleware component can process requests, modify responses, or pass the request to the next middleware in the pipeline. Common middleware includes authentication, logging, and error handling.

---

8. Performance and Optimization

Performance Improvements in .NET Core and .NET

- **Question:** What performance improvements does .NET Core offer over .NET Framework?
  Answer: .NET Core offers significant performance improvements over the .NET Framework, including better memory management, faster garbage collection, and optimizations for JIT (Just-In-Time) compilation. It also introduces new features like span-based types (`Span<T>` , `Memory<T>`) for high-performance memory manipulation.

Kestrel Web Server

- **Question:** What is the Kestrel web server in ASP.NET Core, and why is it important?
  Answer: Kestrel is a cross-platform, high-performance web server used by ASP.NET Core applications. It is designed to be lightweight and fast, suitable for running applications in production environments. Kestrel can be used as a standalone web server or in conjunction with IIS or Nginx.

---

9. Dependency Injection (DI) in .NET Core

Built-in Dependency Injection

- **Question:** How is dependency injection (DI) implemented in .NET Core?
  Answer: .NET Core has built-in support for dependency injection (DI). Services are registered in the `Startup.ConfigureServices()` method and injected into controllers, classes, or services through constructor injection. Services can be registered with different lifetimes (`Transient`, `Scoped`, `Singleton`).

Service Lifetimes

- **Question:** What are the different service lifetimes available in .NET Core DI?
  Answer: The three service lifetimes in .NET Core are:
  - **Transient**: A new instance of the service is created every time it is requested.
  - **Scoped**: A new instance of the service is created for each request.
  - **Singleton**: A single instance of the service is created and shared across the application.

---

10. Asynchronous Programming

Async and Await in .NET

- **Question:** How do `async` and `await` work in .NET and .NET Core?
  Answer: `async` and `await` enable asynchronous programming in .NET by allowing methods to run without blocking the calling thread. This is particularly useful for I/O-bound operations like reading files, database access, or web service calls, improving application responsiveness and scalability.

:

Task-Based Asynchronous Pattern (TAP)

- **Question:** What is the Task-Based Asynchronous Pattern (TAP) in .NET?
  Answer: The Task-Based Asynchronous Pattern (TAP) is the preferred model for asynchronous programming in .NET. It uses `Task` and `Task<T>` types to represent asynchronous operations. TAP simplifies writing asynchronous code with `async` and `await` keywords, enabling better exception handling and code readability. Tasks are executed asynchronously without blocking the main thread.

---

11. Security in .NET and .NET Core

Authentication and Authorization

- **Question:** How is authentication and authorization handled in ASP.NET Core?
  Answer: ASP.NET Core uses middleware for authentication and authorization. Authentication schemes like JWT Bearer tokens, cookies, and OAuth are handled via middleware components. Authorization policies and roles are defined using the `[Authorize]` attribute or custom policies, which control access to different parts of the application.

Data Protection APIs

- **Question:** What is the Data Protection API in ASP.NET Core?
  Answer: The Data Protection API in ASP.NET Core is used to securely store sensitive data, such as authentication tokens, passwords, or personal information. It encrypts data and provides built-in mechanisms to manage encryption keys and securely store them across different environments.

---

12. **Microservices and Cloud Development**

Microservices Architecture with .NET Core

- **Question:** How is .NET Core suited for microservices architecture?
  Answer: .NET Core is ideal for microservices architecture due to its modularity, lightweight footprint, cross-platform capabilities, and fast startup times. It supports containerization with Docker and integrates well with cloud platforms like Azure, allowing services to be deployed and scaled independently.

Azure Integration with .NET Core

- **Question:** How is .NET Core integrated with Microsoft Azure for cloud development?
  Answer: .NET Core has built-in integration with Microsoft Azure, supporting services like Azure App Service, Azure Functions, Azure Storage, and Azure Cosmos DB. Applications built on .NET Core can easily be deployed and managed in the Azure cloud environment, leveraging Azure's infrastructure for scalability, security, and monitoring.

---

13. **Cross-Platform Tools and Development**

Developing on Linux and macOS

- **Question:** How do you develop .NET Core applications on Linux and macOS?
  Answer: .NET Core is cross-platform, allowing development on Linux and macOS using IDEs like Visual Studio Code, JetBrains Rider, or command-line tools like the .NET Core CLI (`dotnet`). The same application code can be built and executed across different platforms without modification.

Docker and Kubernetes with .NET Core

- **Question:** How do Docker and Kubernetes work with .NET Core applications?
  Answer: Docker allows .NET Core applications to be containerized and run consistently across different environments. Kubernetes can be used to orchestrate and manage the deployment of .NET Core applications in a microservices architecture, providing features like scaling, load balancing, and automatic failover.

---

14. **Versioning and Compatibility**

.NET Versioning

- **Question:** How does versioning work in .NET Core and .NET?
  Answer: .NET Core and .NET follow a versioning system where new major versions (e.g., .NET 5, .NET 6) introduce new features and capabilities while maintaining backward compatibility. Each

release includes improvements in performance, new APIs, and sometimes breaking changes that require code adjustments.

Long-Term Support (LTS)

- **Question:** What is the Long-Term Support (LTS) policy for .NET Core and .NET?
  Answer: LTS versions of .NET Core and .NET are supported for at least three years, receiving regular security updates and critical fixes. LTS versions are recommended for production environments, while Current releases may receive feature updates but have a shorter support lifecycle.

---

15. **Frameworks and Libraries in .NET**

.NET MAUI (Multi-platform App UI)

- **Question:** What is .NET MAUI, and how is it used for cross-platform app development?
  Answer: .NET MAUI (Multi-platform App UI) is the evolution of Xamarin.Forms, enabling developers to build cross-platform applications that run on Windows, macOS, iOS, and Android with a single codebase. It allows developers to share UI, business logic, and services across platforms while maintaining platform-specific customizations.

Blazor

- **Question:** What is Blazor in .NET, and how does it enable web development?
  Answer: Blazor is a web framework in .NET that allows developers to build interactive web applications using C# instead of JavaScript. It runs on both the client (Blazor WebAssembly) and the server (Blazor Server). Blazor WebAssembly enables .NET code to run directly in the browser using WebAssembly, while Blazor Server hosts the application logic on the server.

---

16. **Deployment Strategies**

Deployment with .NET Core

- **Question:** What are the different deployment strategies for .NET Core applications?
  Answer: .NET Core applications can be deployed using two primary strategies:
  - **Framework-Dependent Deployment (FDD):** The application is dependent on the presence of .NET Core runtime on the host machine.
  - **Self-Contained Deployment (SCD):** The application includes the .NET Core runtime, allowing it to run independently of the host environment. This is useful for scenarios where the target system does not have .NET Core installed.

Running .NET Applications in Azure

- **Question:** How do you deploy .NET applications to Azure?
  Answer: .NET applications can be deployed to Azure using various methods, such as:
  - **Azure App Service**: A platform-as-a-service (PaaS) offering to host web applications.
  - **Azure Kubernetes Service (AKS)**: For containerized .NET Core applications in Docker.
  - **Azure Functions**: For serverless functions written in .NET. Deployment tools like Azure DevOps and GitHub Actions can automate the build and deployment process.

---

17. **Migration from .NET Framework to .NET Core**

Key Considerations for Migration

- **Question:** What are the key considerations when migrating an application from .NET Framework to .NET Core?
  Answer: When migrating from .NET Framework to .NET Core, you need to consider:
  - **Platform dependencies**: Ensure any Windows-specific APIs are replaced with cross-platform alternatives.
  - **Third-party libraries**: Ensure dependencies are compatible with .NET Core.
  - **App architecture**: Refactor the application to use .NET Core features such as Dependency Injection and ASP.NET Core.
  - **Testing and validation**: Comprehensive testing to ensure feature parity and performance improvements.

Migration Tools

- **Question:** Are there tools available to assist with migrating from .NET Framework to .NET Core?
  Answer: Yes, tools like the .NET Portability Analyzer and the .NET Upgrade Assistant can help identify APIs that need to be updated and provide step-by-step guidance on migrating from .NET Framework to .NET Core.

---

18. **Best Practices for .NET Core Development**

Code Optimization

- **Question:** What are some best practices for optimizing .NET Core applications?
  Answer: Best practices for optimizing .NET Core applications include:
  - **Use** `AsNoTracking()` for read-only queries to improve performance.
  - **Avoid synchronous I/O** in asynchronous code.
  - **Use** `Span<T>` **and** `Memory<T>` for high-performance memory access.
  - **Leverage caching** to reduce database calls.
  - **Optimize middleware** by placing lightweight middleware early in the pipeline.

Logging and Monitoring

- **Question:** How is logging and monitoring handled in .NET Core applications?
  Answer: Logging in .NET Core is built-in and can be configured using the `ILogger` interface. You can log information to different providers such as the console, files, or external logging platforms like Serilog, NLog, or Azure Application Insights. Monitoring can be integrated with Azure Monitor for performance tracking and error reporting.
