ASP.NET Core Interview Preparation for 2 Years of Experience

June 16, 2025

Introduction

This document provides a comprehensive guide for preparing for an ASP.NET Core interview with approximately 2 years of experience. It includes key questions, concise answers, and practical tips to demonstrate your expertise effectively.

1 Key Questions and Answers

1.1 What is ASP.NET Core, and how does it differ from ASP.NET Framework?

Answer: ASP.NET Core is a cross-platform, open-source framework for building modern web applications and APIs. Unlike ASP.NET Framework, which is Windows-only and tightly coupled to .NET Framework, ASP.NET Core:

- Runs on .NET Core or .NET 5+ (cross-platform: Windows, Linux, macOS).
- Offers a modular architecture with minimal dependencies.
- Supports microservices and cloud-based applications.
- Uses a unified pipeline for MVC and Razor Pages.
- Provides better performance due to lightweight design.

Tip: Mention a project where you leveraged ASP.NET Cores cross-platform capability or performance benefits.

1.2 What is the role of the Program.cs and Startup.cs files in ASP.NET Core?

Answer:

- **Program.cs**: Entry point of the application. Configures the host and web server (e.g., Kestrel) using `IHostBuilder`.

- **Startup.cs** (pre-.NET 6): Configures services (dependency injection) and the request pipeline (middleware). In .NET 6+, `Startup.cs` functionality is merged into `Program.cs`.

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllers();
var app = builder.Build();
app.UseRouting();
app.MapControllers();
app.Run();
```

Tip: Explain how you configured middleware or services in a project.

1.3 What is middleware in ASP.NET Core, and how do you create custom middleware?

Answer: Middleware is software that processes HTTP requests and responses in a pipeline. Each middleware component can handle requests, modify responses, or pass control to the next component.

Custom Middleware Example:

```
public class CustomMiddleware
{
    private readonly RequestDelegate _next;
    public CustomMiddleware(RequestDelegate next)
    {
        _next = next;
    }
    public async Task InvokeAsync(HttpContext context)
    {
        // Pre-processing
        await context.Response.WriteAsync("Custom␣Middleware␣
           Start\n");
        await _next(context);
        // Post-processing
        await context.Response.WriteAsync("Custom␣Middleware␣End\
           n");
    }
}
// Extension method
public static class CustomMiddlewareExtensions
{
    public static IApplicationBuilder UseCustomMiddleware(this
       IApplicationBuilder builder)
    {
        return builder.UseMiddleware<CustomMiddleware>();
    }
}
// Usage in Program.cs
app.UseCustomMiddleware();
```

Tip: Share a scenario where you wrote custom middleware, e.g., for logging or authentication.

1.4 How does Dependency Injection (DI) work in ASP.NET Core?

Answer: DI in ASP.NET Core manages dependencies by injecting services into controllers, services, or other classes. Services are registered in `Program.cs` with lifetimes: `Transient`, `Scoped`, or `Singleton`.

```
builder.Services.AddScoped<IUserService, UserService>();
public class UserController : ControllerBase
{
    private readonly IUserService _userService;
    public UserController(IUserService userService)
    {
        _userService = userService;
    }
}
```

Tip: Explain the difference between lifetimes and a case where you chose one over another.

1.5 What is the difference between Transient, Scoped, and Singleton service lifetimes?

Answer:

- **Transient**: New instance per request. Suitable for lightweight services.
- **Scoped**: Same instance within a request scope (e.g., HTTP request). Ideal for database contexts.
- **Singleton**: Same instance for the application lifetime. Used for shared resources.

Tip: Mention a project where you used `Scoped` for Entity Framework or `Singleton` for caching.

1.6 What is Entity Framework Core, and how do you use it in ASP.NET Core?

Answer: Entity Framework Core (EF Core) is an ORM for data access. It maps .NET classes to database tables and supports LINQ queries.

```
public class AppDbContext : DbContext
{
    public DbSet<User> Users { get; set; }
    public AppDbContext(DbContextOptions<AppDbContext> options) :
        base(options) { }
}
// Register in Program.cs
builder.Services.AddDbContext<AppDbContext>(options =>
```

```
    options.UseSqlServer(connectionString));
// Usage
public class UserService
{
    private readonly AppDbContext _context;
    public UserService(AppDbContext context)
    {
        _context = context;
    }
    public async Task<List<User>> GetUsersAsync()
    {
        return await _context.Users.ToListAsync();
    }
}
```

Tip: Discuss migrations or a complex query you wrote.

1.7 How do you create a RESTful API in ASP.NET Core?

Answer: Use controllers with `ControllerBase` and HTTP attributes (`[HttpGet]`, `[HttpPost]`, etc.).

```
[ApiController]
[Route("api/[controller]")]
public class UsersController : ControllerBase
{
    private readonly IUserService _userService;
    public UsersController(IUserService userService)
    {
        _userService = userService;
    }
    [HttpGet]
    public async Task<IActionResult> GetUsers()
    {
        var users = await _userService.GetUsersAsync();
        return Ok(users);
    }
    [HttpPost]
    public async Task<IActionResult> CreateUser(User user)
    {
        await _userService.CreateUserAsync(user);
        return CreatedAtAction(nameof(GetUsers), new { id = user.
            Id }, user);
    }
}
```

Tip: Mention versioning or Swagger integration if used.

1.8 What is the purpose of appsettings.json, and how do you access its values?

Answer: `appsettings.json` stores configuration settings (e.g., connection strings). Access values using `IConfiguration`.

```
// appsettings.json
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=localhost;Database=MyDb;..."
  }
}
// Access in Program.cs
var connectionString = builder.Configuration.GetConnectionString(
   "DefaultConnection");
```

Tip: Share how you managed environment-specific configurations.

1.9 How do you handle authentication and authorization in ASP.NET Core?

Answer: Use `AddAuthentication` and `AddAuthorization` for authentication (e.g., JWT) and role-based or policy-based authorization.

```
builder.Services.AddAuthentication(JwtBearerDefaults.
   AuthenticationScheme)
    .AddJwtBearer(options => { /* Configure JWT */ });
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("AdminOnly", policy => policy.RequireRole("
       Admin"));
});
// Usage
[Authorize(Policy = "AdminOnly")]
public class AdminController : ControllerBase { }
```

Tip: Discuss implementing JWT or OAuth in a project.

1.10 What is the difference between MVC and Razor Pages in ASP.NET Core?

Answer:

- **MVC**: Separates concerns (Model, View, Controller). Ideal for APIs and complex apps.
- **Razor Pages**: Page-based model combining view and controller logic. Suitable for simpler apps.

Tip: Explain when you chose one over the other.

1.11 How do you handle exceptions globally in ASP.NET Core?

Answer: Use middleware or `ExceptionFilter` for global exception handling.

```
app.UseExceptionHandler(errorApp =>
{
    errorApp.Run(async context =>
    {
        var error = context.Features.Get<IExceptionHandlerFeature
            >();
        await context.Response.WriteAsJsonAsync(new { Error =
            error?.Error.Message });
    });
});
```

Tip: Share a custom error response you implemented.

1.12 What is CORS, and how do you configure it in ASP.NET Core?

Answer: CORS (Cross-Origin Resource Sharing) allows cross-domain requests. Configure using `AddCors`.

```
builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowSpecificOrigin",
        builder => builder.WithOrigins("https://example.com").
            AllowAnyMethod().AllowAnyHeader());
});
app.UseCors("AllowSpecificOrigin");
```

Tip: Mention a CORS issue you resolved.

1.13 How do you optimize ASP.NET Core application performance?

Answer: Techniques:

- Use asynchronous programming (`async/await`).
- Enable response caching.
- Minimize database queries with EF Core optimizations (e.g., `AsNoTracking`).
- Use compression (`UseResponseCompression`).
- Optimize middleware order.

Tip: Share a performance improvement you achieved.

1.14 What is Swagger, and how do you integrate it in ASP.NET Core?

Answer: Swagger (OpenAPI) documents APIs. Integrate using `Swashbuckle`.

```
builder.Services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo { Title = "My API",
       Version = "v1" });
});
app.UseSwagger();
app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json
   ", "My API V1"));
```

Tip: Mention adding authentication to Swagger.

1.15 How do you implement unit testing in ASP.NET Core?

Answer: Use xUnit or NUnit with Moq for mocking dependencies. Test controllers and services using `TestServer` or in-memory databases.

```
public class UserServiceTests
{
    [Fact]
    public async Task GetUsers_ReturnsUsers()
    {
        var mockRepo = new Mock<IUserRepository>();
        mockRepo.Setup(repo => repo.GetUsersAsync()).ReturnsAsync
           (new List<User>());
        var service = new UserService(mockRepo.Object);
        var result = await service.GetUsersAsync();
        Assert.Empty(result);
    }
}
```

Tip: Discuss a test case you wrote.

2 Preparation Tips

- **Know Your Projects**: Discuss challenges (e.g., optimizing EF queries).
- **Code Examples**: Practice writing APIs or middleware.
- **Async Programming**: Master `async/await` patterns.
- **Performance**: Study caching and EF optimizations.
- **Mock Interviews**: Practice explaining solutions.
- **Stay Updated**: Learn about .NET 9 features.
- **Ask Questions**: Inquire about teams architecture or testing practices.

3 Additional Questions

- **Hosted Services**: Background tasks using `IHostedService`.
- **SignalR**: Real-time communication in ASP.NET Core.
- **gRPC**: High-performance RPC framework in .NET.
- **Health Checks**: Monitor app health with `AddHealthChecks`.