.NET Core

ASP.NET Core Architecture – Overview

Key Exploration Points

- Layered architecture concept
- Separation of concerns
- Request–response pipeline
- Middleware-based design
- Scalability and maintainability

EXPLANATION

ASP.NET Core follows a layered and modular architecture designed for modern web applications. Each layer has a clear responsibility, which improves maintainability and testability. The framework processes requests through a middleware pipeline before reaching controllers. This design supports scalability and performance for enterprise systems. Understanding the architecture helps freshers debug issues and extend applications confidently.

Example Flow

```
Client → Middleware → Controller → Service → Response
```

ASP.NET Core Request Pipeline

Key Exploration Points

- Middleware execution order
- Request handling flow
- Response generation
- Cross-cutting concerns
- Custom middleware

EXPLANATION

Every HTTP request passes through a sequence of middleware components. Middleware handles cross-cutting concerns like logging, authentication, and exception handling. The order of middleware is critical because it affects application behavior. Developers can add custom middleware when required. Proper pipeline configuration is essential for stable APIs.

Example Pipeline Configuration

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

Program.cs – Role and Responsibilities

Key Exploration Points

- Application startup
- Service registration
- Middleware configuration
- Environment awareness
- Hosting model

EXPLANATION

Program.cs is the entry point of an ASP.NET Core application. It defines how the application starts, what services are available, and how requests are handled. With the minimal hosting model, all startup logic is consolidated in one place. Developers configure middleware and dependency injection here. A clear understanding of Program.cs is critical for API configuration and troubleshooting.

Startup Logic

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
```

Minimal Hosting Model (Modern ASP.NET Core)

Key Exploration Points

- Introduced in .NET 6+
- Reduced boilerplate code
- Simplified startup
- Readability improvement
- Faster onboarding

EXPLANATION

The minimal hosting model removes the need for a separate Startup class. Configuration and service setup happen in Program.cs itself. This reduces complexity and improves readability for new developers. It allows freshers to quickly understand application flow. Modern enterprise projects follow this approach.

Minimal Service Registration

```
builder.Services.AddControllers();
```

Dependency Injection – Why It Matters

Key Exploration Points

- Loose coupling
- Testability
- Maintainability
- Replaceable implementations
- Industry-standard practice

EXPLANATION

Dependency Injection (DI) allows objects to receive their dependencies instead of creating them. This reduces tight coupling between components. DI improves testability by allowing mock implementations. It also supports maintainability and future changes. ASP.NET Core provides built-in DI, which is heavily used in real projects.

Service Registration Example

```
builder.Services.AddScoped();
```

Service Lifetimes in Dependency Injection

Key Exploration Points

- Transient services
- Scoped services
- Singleton services
- Request-based lifecycle
- Performance considerations

EXPLANATION

Service lifetimes define how long an object instance lives. Transient services are created every time they are requested. Scoped services are created once per HTTP request. Singleton services are shared across the application lifetime. Choosing the wrong lifetime can cause performance or data issues. Freshers must understand lifetimes clearly.

Singleton Lifecycle Example

```
builder.Services.AddSingleton();
```

Controllers – Role in Web API

Key Exploration Points

- Request handling
- HTTP endpoints
- Input validation
- Delegating business logic
- API responsibility

EXPLANATION

Controllers act as entry points for HTTP requests in Web APIs. They receive input, validate it, and delegate business logic to services. Controllers should remain thin and not contain business logic. This improves readability and testability. In enterprise projects, controllers are heavily reviewed for clean design.

Controller Definition

```
[ApiController]
public class UsersController : ControllerBase { }
```

MVC Pattern – Conceptual Understanding

Key Exploration Points

- Model, View, and Controller
- Separation of concerns
- Web API vs MVC UI
- Architecture consistency
- Data flow responsibility

EXPLANATION

MVC stands for Model, View, and Controller. In Web APIs, the View layer is usually absent, and responses are JSON. Models represent data, controllers handle requests, and services contain logic. This separation improves maintainability. Understanding MVC concepts helps freshers adapt to both API and UI projects.

MVC Data Flow

```
Controller → Service → Model → Response
```

REST – Core Principles

Key Exploration Points

- Stateless communication
- Resource-based design
- Uniform interface
- Client-server separation
- Scalability

EXPLANATION

REST is an architectural style used for building scalable APIs. Each request is stateless, meaning it contains all necessary information. APIs expose resources rather than actions. Uniform interfaces ensure consistency. RESTful design is essential for modern integrations.

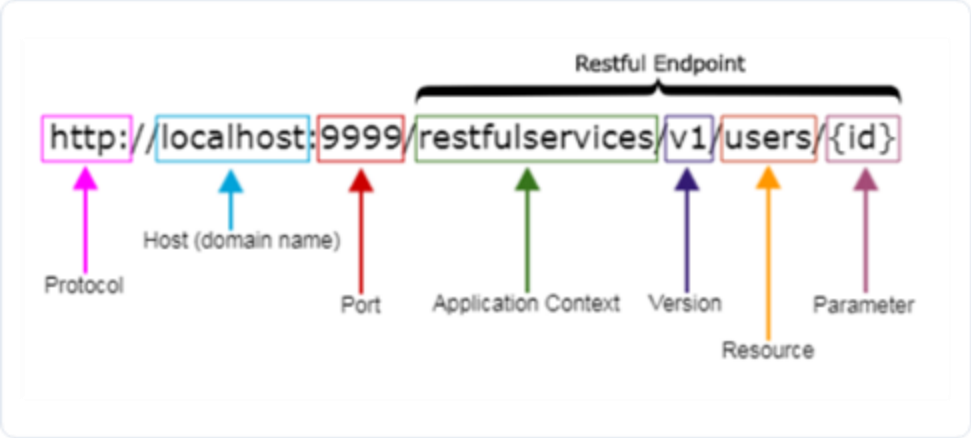
Resource Example

```
GET /api/users
```

RESTful Resource Naming Best Practices

Key Exploration Points

- Noun-based endpoints
- Plural resource names
- Avoid verbs in URLs
- Consistency patterns
- Versioning awareness



EXPLANATION

REST APIs should use nouns to represent resources. Plural naming improves clarity. Verbs are represented using HTTP methods instead of URLs. Consistent naming improves API usability. Versioning helps manage changes safely.

Proper Resource POST

```
POST /api/orders
```

HTTP Verbs – Purpose and Usage

Key Exploration Points

- GET: Data retrieval
- POST: Resource creation
- PUT: Data updates
- DELETE: Removal
- PATCH: Partial updates

EXPLANATION

HTTP verbs define the action to be performed on a resource. GET retrieves data, POST creates new resources, PUT updates existing data, and DELETE removes resources. PATCH is used for partial updates. Correct usage improves API clarity and client understanding. Incorrect verb usage leads to poor API design.

Action Attribute

```
[HttpGet]
public IActionResult GetUsers()
```

HTTP Status Codes – Communicating Results

Key Exploration Points

- 200-series: Success
- 400-series: Client Errors
- 500-series: Server Errors
- Meaningful responses
- API reliability

EXPLANATION

Status codes communicate the result of an API request. Success responses use 200-series codes. Client errors use 400-series codes, and server issues use 500-series codes. Proper status codes improve debugging and integration. APIs must return accurate status codes for reliability.

Success Response

```
return Ok(users);
```

Routing – Mapping Requests to Actions

Key Exploration Points

- Attribute routing
- Route templates
- Parameter binding
- Route constraints
- Versioning support

EXPLANATION

Routing determines how incoming requests map to controller actions. Attribute routing is commonly used in Web APIs for clarity. Route parameters allow dynamic values. Constraints improve validation. Proper routing ensures predictable API behavior.

Parameter Mapping

```
[HttpGet("{id}")]
public IActionResult GetUser(int id)
```

Route Best Practices

Key Exploration Points

- Clear route structure
- Avoid ambiguity
- Consistent patterns
- Versioned routes
- Maintainability

EXPLANATION

Routes should be simple and predictable. Ambiguous routes cause runtime errors. Consistency across controllers improves readability. Versioned routes help manage breaking changes. Clean routing improves API usability.

Versioned Route Example

```
/api/v1/users/{id}
```

Swagger – Purpose and Benefits

Key Exploration Points

- API documentation
- Interactive testing
- Consumer visibility
- Faster onboarding
- Industry standard

EXPLANATION

Swagger provides interactive API documentation. It allows developers and testers to explore APIs without external tools. Swagger improves onboarding for new team members. It acts as a live contract between API and consumers. Most enterprise APIs expose Swagger.

Generator Registration

```
builder.Services.AddSwaggerGen();
```

Swagger Best Practices

Key Exploration Points

- Enable in development
- Secure in production
- Use descriptive annotations
- Clear API descriptions
- Versioned documentation

EXPLANATION

Swagger should be enabled by default in development environments. In production, it must be secured or disabled. Proper annotations improve documentation quality. Clear descriptions reduce dependency on verbal explanations. Versioning ensures backward compatibility.

Middleware Setup

```
app.UseSwagger();
app.UseSwaggerUI();
```

Thank You

Any Questions?