1. 3-Tier System Architecture

- **Overview**: A layered software architecture where the application is divided into three layers: Presentation, Business Logic, and Data.
- **Definition**: It separates concerns into distinct layers that improve maintainability, scalability, and manageability.
- **Importance**: Provides clean separation of concerns, making the system modular, maintainable, and scalable.
- **Real-world Use Case**: Enterprise applications, like online banking systems, CRM systems.

Code Snippet:

```
 // Example of a 3-tier architecture in .NET
// Presentation Layer
public class UserController : Controller
{
    private readonly IUserService _userService;
    public UserController(IUserService userService)
    {
        _userService = userService;
    }
    public IActionResult Index()
    {
        var users = _userService.GetUsers();
        return View(users);
    }
}

// Business Logic Layer (BLL)
public class UserService : IUserService
{
    private readonly IUserRepository _userRepo;
    public UserService(IUserRepository userRepo)
    {
        _userRepo = userRepo;
    }
    public IEnumerable<User> GetUsers()
    {
        return _userRepo.GetAll();
    }
}

// Data Access Layer (DAL)
public class UserRepository : IUserRepository
{
    private readonly DbContext _context;
    public UserRepository(DbContext context)
    {
        _context = context;
    }
    public IEnumerable<User> GetAll()
    {
        return _context.Users.ToList();
    }
}
```

-

2. Application Server Layered Architecture

- **Overview**: Involves multiple tiers such as Presentation Layer, Business Logic Layer (BLL), Data Access Layer (DAL), and Infrastructure Layer.
- **Definition**: Different responsibilities and functionalities are encapsulated in separate layers.
- **Importance**: Simplifies management, testing, and scaling of software.
- **Real-world Use Case**: Large-scale enterprise applications like e-commerce platforms.

Code Snippet:

```
 // Presentation Layer (e.g., MVC)
// Business Logic Layer
// Data Access Layer
```

-

3. Data Transfer Objects (DTO)

- **Overview**: DTOs are simple objects used to transfer data between layers or over a network.
- **Definition**: A design pattern used to encapsulate data for transfer and reduce the number of method calls.
- **Importance**: Helps in decoupling, optimizing data transfer, and improving performance.
- **Real-world Use Case**: Sending data over an API, passing large data sets between service layers.

Code Snippet:

```
public class UserDTO
{
   public int Id { get; set; }
   public string Name { get; set; }
   public string Email { get; set; }
}
```

-

4. Repository Pattern in .NET and Generic Repository

- **Overview**: The repository pattern provides a way to encapsulate data access logic.
- **Definition**: A repository abstracts the data layer, providing a collection-like interface to access data.
- **Importance**: Simplifies data access and helps to decouple business logic from data access logic.
- **Real-world Use Case**: Managing database operations in large applications.

Code Snippet:

```
public interface IGenericRepository<T> where T : class
{
   IEnumerable<T> GetAll();
   T GetById(int id);
```

```
    void Add(T entity);
    void Update(T entity);
    void Delete(int id);
}

public class GenericRepository<T> : IGenericRepository<T> where T : class
{
    private readonly DbContext _context;
    public GenericRepository(DbContext context)
    {
        _context = context;
    }
    public IEnumerable<T> GetAll() { return _context.Set<T>().ToList(); }
    // Other methods implementation...
}
```

-

5. Unit of Work Pattern in .NET

- **Overview**: The Unit of Work pattern ensures that all data operations within a business transaction are completed successfully before committing.
- **Definition**: A design pattern that manages transactions and coordinates the work of multiple repositories.
- **Importance**: Helps ensure that data integrity is maintained and operations are consistent.
- **Real-world Use Case**: E-commerce checkout systems where multiple repositories are involved.

Code Snippet:

```
 public interface IUnitOfWork
{
    IGenericRepository<User> Users { get; }
    IGenericRepository<Order> Orders { get; }
    int Complete();
}

public class UnitOfWork : IUnitOfWork
```

```
{
    private readonly DbContext _context;
    public IGenericRepository<User> Users { get; }
    public IGenericRepository<Order> Orders { get; }

    public UnitOfWork(DbContext context)
    {
        _context = context;
        Users = new GenericRepository<User>(_context);
        Orders = new GenericRepository<Order>(_context);
    }

    public int Complete() => _context.SaveChanges();
}
```

-

6. Dependency Injection & Inversion of Control

- **Overview**: Dependency Injection (DI) allows an object to receive its dependencies from an external source rather than creating them internally.
- **Definition**: A design principle that allows decoupling by passing dependencies into classes or methods.
- **Importance**: Increases flexibility, modularity, and testability.
- **Real-world Use Case**: Injecting services like logging, data access, etc.

Code Snippet:

```
 // Registering DI in Startup.cs
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<IUserService, UserService>();
}

// Usage
public class HomeController : Controller
{
    private readonly IUserService _userService;
    public HomeController(IUserService userService)
    {
        _userService = userService;
    }
}
```

-

7. Middleware in .NET and Middleware Pipeline

- **Overview**: Middleware in .NET is software that processes HTTP requests and responses in a pipeline.
- **Definition**: A component that inspects and/or modifies requests before passing them to the next component.
- **Importance**: Customizes request handling, adding cross-cutting concerns like authentication, logging, etc.
- **Real-world Use Case**: Authentication middleware, logging middleware.

Code Snippet:

```
public class LoggingMiddleware
{
    private readonly RequestDelegate _next;
    public LoggingMiddleware(RequestDelegate next)
    {
        _next = next;
    }
    public async Task InvokeAsync(HttpContext context)
    {
        Console.WriteLine("Request received");
        await _next(context);
    }
}
```

-

8. AppSettings.json

- **Overview**: A configuration file used in ASP.NET Core to store application settings.
- **Definition**: JSON-based file for storing settings like database connection strings, API keys.

- **Importance**: Centralizes configuration and simplifies management.
- **Real-world Use Case**: Storing database connection strings, API keys.

Code Snippet:

```
 {
  "ConnectionStrings": {
    "DefaultConnection": "Server=myserver;Database=mydb;User
Id=myuser;Password=mypassword;"
  }
}
```

-

9. REST API and HTTP Protocol

- **Overview**: REST (Representational State Transfer) is an architectural style for web services that uses HTTP as a protocol.
- **Definition**: A stateless, client-server communication style for accessing resources.
- **Importance**: Allows scalable and flexible communication between clients and servers.
- **Real-world Use Case**: Web services like social media APIs, payment gateways.

Code Snippet:

```
 [HttpGet]
public ActionResult<IEnumerable<User>> GetUsers()
{
    return _userService.GetUsers().ToList();
}
```

-

10. Swagger API Documentation

- **Overview**: Swagger is a tool that helps document REST APIs.
- **Definition**: Automatically generates API documentation that is interactive and user-friendly.
- **Importance**: Simplifies API integration and testing.
- **Real-world Use Case**: API testing, documentation for public APIs.

Code Snippet:

```
 public void ConfigureServices(IServiceCollection services)
{
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title = "My API", Version = "v1" });
    });
}
```

-

11. API Versioning

- **Overview**: A technique to manage different versions of an API.
- **Definition**: Ensures backward compatibility and smooth transitions for changes in the API.
- **Importance**: Allows clients to continue using old versions while supporting newer features.
- **Real-world Use Case**: Managing breaking changes in an API over time.

Code Snippet:

```
 services.AddApiVersioning(options =>
{
    options.ReportApiVersions = true;
    options.AssumeDefaultVersionWhenUn
```

Here's a detailed breakdown for the remaining topics:

12. Swagger API Documentation

- **Overview**: Swagger is a framework used to generate interactive documentation for REST APIs. It helps developers explore and test APIs directly from the documentation.
- **Definition**: Swagger generates a user-friendly interface that provides details about API endpoints, parameters, responses, and authentication.
- **Importance**: Makes it easier for developers and clients to understand, test, and integrate with APIs.

- **Real-world Use Case**: Common in APIs for mobile apps, web applications, and third-party integrations where documentation needs to be accessible and interactive.

Code Snippet:

```
 // In Startup.cs
public void ConfigureServices(IServiceCollection services)
{
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title = "My API", Version = "v1" });
    });
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseSwagger();
        app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API v1"));
    }
}
```

-

13. API Versioning

- **Overview**: API versioning ensures that different versions of an API can exist simultaneously, making it easier to manage breaking changes and new features.
- **Definition**: A method to handle different versions of an API without affecting existing clients.
- **Importance**: It allows for backward compatibility while introducing new features and preventing disruptions for clients.
- **Real-world Use Case**: When adding new features to an API that may change existing behavior, versioning helps clients continue using the old version until they are ready to migrate.

Code Snippet:

```
 // In Startup.cs for versioning setup
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddApiVersioning(options =>
    {
        options.AssumeDefaultVersionWhenUnspecified = true;
        options.ReportApiVersions = true;
    });
}

[ApiVersion("1.0")]
[Route("api/v{version:apiVersion}/[controller]")]
public class MyApiController : ControllerBase
{
    // Action methods here
}
```

-

14. Routing in .NET

- **Overview**: Routing in .NET is the process of mapping incoming requests to the appropriate controller action.
- **Definition**: It is a mechanism used to define URL patterns and associate them with specific actions in controllers.
- **Importance**: Provides flexibility in defining custom URLs and handling different HTTP methods, making the application more RESTful and user-friendly.
- **Real-world Use Case**: RESTful APIs, MVC applications where the URL structure is important for user navigation.

Code Snippet:

```
 // In Startup.cs
public void Configure(IApplicationBuilder app)
{
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}

// In Controller
[Route("api/[controller]")]
public class MyController : ControllerBase
{
    [HttpGet("get/{id}")]
    public IActionResult Get(int id)
    {
        return Ok($"Item {id}");
    }
}
```

-

15. Model Binding and Validation

- **Overview**: Model binding in .NET refers to the process of mapping incoming HTTP request data (such as JSON or form data) to a C# object. Validation ensures that the data conforms to expected formats and rules.
- **Definition**: Model binding automatically binds HTTP request data to action method parameters or model properties. Validation ensures that the data satisfies specified rules before it's processed.
- **Importance**: Simplifies the process of handling input from users, and validation ensures data integrity and security by rejecting invalid inputs.
- **Real-world Use Case**: User input forms, API endpoints where client data needs to be validated.

Code Snippet:

```
 public class CreateUserModel
{
    [Required]
    [StringLength(100, MinimumLength = 3)]
    public string Name { get; set; }
    [Required]
    [EmailAddress]
    public string Email { get; set; }
}

[HttpPost]
public IActionResult CreateUser([FromBody] CreateUserModel model)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    return Ok("User created");
}
```