Here's a comprehensive set of .NET Core MVC interview questions and answers, covering fundamental concepts to more advanced topics.

---

1. Core MVC Concepts

1. What is MVC and how does it apply to .NET Core?

MVC stands for **Model-View-Controller**. It's an architectural pattern that separates an application into three main components:

- **Model:** Represents the application's data, business logic, and rules. It's independent of the user interface.
- **View:** Displays the data from the Model to the user and handles user interface logic. It's responsible for the presentation.
- **Controller:** Handles user input, interacts with the Model, and selects the appropriate View to display. It acts as an intermediary.

In .NET Core, MVC is a framework for building web applications that follow this pattern. It promotes separation of concerns, testability, and maintainability.

2. What are the key advantages of using ASP.NET Core MVC?

- **Separation of Concerns:** Clearly separates data, UI, and logic, making the codebase more organized and easier to manage.
- **Testability:** Decoupled components are easier to unit test independently.
- **Routing:** Flexible and powerful URL routing system.
- **Open-Source & Cross-Platform:** Runs on Windows, macOS, and Linux, and its source code is publicly available.
- **Performance:** Designed for high performance and scalability.
- **Dependency Injection (DI) Built-in:** Native support for DI, promoting loose coupling.
- **Modular and Lightweight:** You only include the components you need, leading to smaller application footprints.
- **Integration with Modern Client-Side Frameworks:** Works well with JavaScript frameworks like Angular, React, and Vue.js.

3. Explain the lifecycle of an ASP.NET Core MVC request.

The lifecycle involves several stages:

1. **Request Reception:** Kestrel (or IIS/Nginx acting as a reverse proxy) receives the HTTP request.
2. **Middleware Pipeline:** The request goes through a series of middleware components configured in `Program.cs` (or `Startup.cs` in older versions). Each middleware can process the request, short-circuit the pipeline, or pass it to the next. Common middleware includes routing, authentication, authorization, and static files.

3. **Routing:** The routing middleware matches the incoming URL to a defined route, determining which controller and action method should handle the request.
4. **Controller Activation:** An instance of the identified controller is created using dependency injection.
5. **Model Binding:** Data from the HTTP request (e.g., query strings, form data, route data) is mapped to the action method's parameters.
6. **Action Filters:** Action filters (e.g., `OnActionExecuting`) are executed before the action method.
7. **Action Execution:** The controller's action method is executed.
8. **Action Result Execution:** The `IActionResult` returned by the action method (e.g., `ViewResult`, `JsonResult`, `RedirectResult`) is executed.
9. **Result Filters:** Result filters (e.g., `OnResultExecuting`) are executed before the action result.
10. **View Rendering (if applicable):** If a `ViewResult` is returned, the Razor View Engine compiles and renders the `.cshtml` view into HTML.
11. **Response Generation:** The final HTTP response is generated and sent back to the client.

---

2. Controllers and Actions

4. What is a Controller in ASP.NET Core MVC?

A **Controller** is a class that handles incoming HTTP requests. Its primary responsibilities include:

- Receiving input from the user.
- Processing the input by interacting with the Model.
- Deciding which View to render to display the results.

Controllers typically inherit from `Microsoft.AspNetCore.Mvc.Controller` (for MVC applications returning views) or `Microsoft.AspNetCore.Mvc.ControllerBase` (for API controllers).

5. What is an Action Method?

An **Action Method** is a public method within a controller that handles a specific HTTP request. By convention, all public methods in a controller are considered action methods unless decorated with the `[NonAction]` attribute.

6. What are common `IActionResult` types and when would you use them?

`IActionResult` is the base interface for all action results. Common derived types include:

- **`ViewResult`**: Renders a view (e.g., `return View();`). Used for traditional web pages.
- **`JsonResult`**: Returns JSON formatted data (e.g., `return Json(data);`). Used for APIs or AJAX responses.

- **`ContentResult`**: Returns plain text content (e.g., `return Content("Hello World");`).
- **`RedirectResult`**: Redirects the client to a specified URL (e.g., `return Redirect("https://example.com");`).
- **`RedirectToActionResult`**: Redirects to another action within the application (e.g., `return RedirectToAction("Index", "Home");`).
- **`FileResult`**: Returns a file (e.g., `return File(filePath, contentType);`).
- **`NotFoundResult`**: Returns an HTTP 404 Not Found status code (e.g., `return NotFound();`).
- **`BadRequestResult`**: Returns an HTTP 400 Bad Request status code (e.g., `return BadRequest();`).
- **`OkResult`**: Returns an HTTP 200 OK status code (e.g., `return Ok();`).

---

3. Views

7. What is a View in ASP.NET Core MVC?

A **View** is the component responsible for rendering the user interface. In ASP.NET Core MVC, views are typically Razor views (`.cshtml` files) which combine HTML with C# code using Razor syntax. They display data passed from the controller.

8. Explain Razor Syntax.

Razor is a markup syntax for embedding server-side C# code into HTML web pages. It provides a concise and expressive way to generate dynamic HTML content. Key features include:

- `@` symbol to transition from HTML to C# code.
- `@{ ... }` for code blocks.
- `@model` directive to declare the model type for a strongly-typed view.
- `@using` for namespaces.
- `@Html.HelperName(...)` for HTML Helpers.

Example:
HTML
```
@model MyProject.Models.Product

<h1>@Model.Name</h1>
<p>Price: @Model.Price.ToString("C")</p>

@{
    var year = DateTime.Now.Year;
}
<p>&copy; @year My Company</p>
```

9. What are Strongly-Typed Views and why are they beneficial?

A **strongly-typed view** is a view that declares a specific model type using the `@model` directive. This provides:

- **Compile-time type checking:** Errors related to model properties are caught at compile time, not runtime.
- **IntelliSense support:** Visual Studio (or other IDEs) provides auto-completion for model properties, improving development speed and reducing errors.
- **Readability and Maintainability:** The code is clearer about what data it expects.

Example: `@model MyProject.Models.UserProfile`

10. Differentiate between `ViewData`, `ViewBag`, and `TempData`.

These are mechanisms to pass data from a controller to a view (and sometimes between requests):

- **`ViewData`**: A `Dictionary<string, object>` accessible via string keys. It's available only for the **current request**.
  - **Pros:** Type-safe if cast explicitly.
  - **Cons:** Requires explicit casting, which can lead to runtime errors if the type is incorrect.
  - **Example:** `ViewData["Message"] = "Hello";` in controller, `@ViewData["Message"]` in view.
- **`ViewBag`**: A `dynamic` wrapper around `ViewData`. It allows property-style access without explicit casting. Available only for the **current request**.
  - **Pros:** Easier to use with no casting needed.
  - **Cons:** Lack of compile-time type checking, prone to runtime errors due to typos.
  - **Example:** `ViewBag.Message = "Hello";` in controller, `@ViewBag.Message` in view.
- **`TempData`**: A `Dictionary<string, object>` that stores data for a **single subsequent request** (typically used for redirects). Data is read and then cleared.
  - **Pros:** Useful for displaying success/error messages after a redirect.
  - **Cons:** Data persists only for the next request unless explicitly kept.
  - **Example:** `TempData["SuccessMessage"] = "Item added!";` in controller, then check `TempData["SuccessMessage"]` in the view of the redirected action. You can use `TempData.Keep("Key")` or `TempData.Peek("Key")` to retain data for more requests.

11. What are Partial Views and when are they used?

A **Partial View** is a reusable Razor view (`.cshtml` file) that renders a portion of a larger view. It doesn't have its own layout page and is typically used for:

- **Code Reusability:** Avoid duplicating UI elements (e.g., common navigation, sidebar widgets).
- **Modularity:** Breaking down complex views into smaller, manageable components.
- **AJAX Updates:** Updating only a specific part of a page without reloading the entire page.

You can render a partial view using `@Html.Partial("PartialViewName")` or `@Html.RenderPartial("PartialViewName")`.

12. Explain the concept of Layout Pages.

A **Layout Page** (`_Layout.cshtml`) provides a consistent look and feel for all views in an ASP.NET Core MVC application. It contains common HTML elements like header, footer, navigation, and script references. Views then use the `@layout` directive to inherit from the layout page and render their specific content within placeholders defined by `@RenderBody()` and `@RenderSection()`.

---

4. Routing

13. How does Routing work in ASP.NET Core MVC?

Routing is the process of mapping incoming HTTP requests to specific controller actions. In ASP.NET Core MVC, routing is configured in the `Program.cs` file (or `Startup.cs`).

- **Convention-based Routing:** Defines patterns that map URLs to controller actions. The default route typically follows the pattern `{controller}/{action}/{id?}`.
- **Attribute Routing:** Allows you to define routes directly on controller classes and action methods using attributes like `[Route]`, `[HttpGet]`, `[HttpPost]`, etc. This provides more granular control and can make routes more readable.

Example (Program.cs):
C#
```
app.UseRouting(); // Enables routing middleware
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
    // Attribute routes are discovered automatically by MapControllers()
if present
    endpoints.MapControllers();
});
```

14. What is the difference between Convention-based Routing and Attribute Routing?

- **Convention-based Routing:**
  - Defined centrally (e.g., in `Program.cs`).
  - Uses predefined URL patterns to map requests to controllers and actions.
  - Good for consistent URL structures across the application.
  - Less flexible for highly customized URLs.
  - Example: `/{controller}/{action}/{id?}`
- **Attribute Routing:**

- Defined directly on controllers and action methods using attributes.
- Provides explicit control over URL patterns for individual actions.
- Excellent for creating RESTful APIs with clear, descriptive URLs.
- Can become harder to manage for very large numbers of routes if not organized well.
- Example:

C#
```
[Route("products")]
public class ProductsController : Controller
{
    [HttpGet("{id}")]
    public IActionResult GetProduct(int id) { /* ... */ }
}
```

---

5. Model Binding and Validation

15. What is Model Binding?

Model Binding is a mechanism in ASP.NET Core MVC that automatically maps data from HTTP requests (e.g., form fields, query string parameters, route data, uploaded files, JSON body) to parameters of action methods or properties of a model class. This simplifies data retrieval from the request.

16. How do you implement validation in ASP.NET Core MVC?

ASP.NET Core MVC provides powerful validation capabilities, primarily through:

- **Data Annotations:** Attributes like `[Required]`, `[StringLength]`, `[Range]`, `[EmailAddress]`, etc., applied to properties of your model classes.
- **`ModelState.IsValid`:** In your controller actions, you check `ModelState.IsValid` after model binding. If `false`, it means validation errors occurred.
- **Client-Side Validation:** Often facilitated by JavaScript libraries (e.g., jQuery Validation) that work with Data Annotations to provide immediate feedback to the user without a server roundtrip.

Example:
C#
```
public class RegisterViewModel
{
    [Required]
    [EmailAddress]
    public string Email { get; set; }

    [Required]
    [StringLength(100, MinimumLength = 6)]
    [DataType(DataType.Password)]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Compare("Password", ErrorMessage = "The password and confirmation
password do not match.")]
    public string ConfirmPassword { get; set; }
}

public class AccountController : Controller
{
    [HttpPost]
    public IActionResult Register(RegisterViewModel model)
    {
        if (ModelState.IsValid)
        {
            // Process valid model
            return RedirectToAction("RegistrationSuccess");
        }
        // If invalid, return to view with validation errors
        return View(model);
    }
}
```

---

6. Dependency Injection

17. What is Dependency Injection (DI) and why is it important in .NET Core?

Dependency Injection (DI) is a design pattern used to achieve **Inversion of Control (IoC)**. Instead of a class creating its own dependencies, those dependencies are "injected" into it, typically through its constructor.

Importance in .NET Core:

- **Loose Coupling:** Reduces dependencies between classes, making components more independent and reusable.
- **Testability:** Easier to unit test components by mocking or substituting their dependencies.
- **Maintainability:** Changes to a dependency's implementation don't require changes to classes that use it.
- **Scalability:** Facilitates building larger, more complex applications by managing object lifetimes and dependencies.

.NET Core has a built-in DI container, making it a first-class citizen in the framework.

18. Explain the different service lifetimes in .NET Core DI: Singleton, Scoped, and Transient.

When registering services in the DI container (usually in `Program.cs` or `Startup.cs`), you specify their lifetime:

- **Singleton (`AddSingleton`):**
  - A single instance of the service is created and shared across the **entire application lifetime**.
  - Subsequent requests for the service will always receive the same instance.
  - Suitable for services that are stateless or expensive to create, and whose state can be shared globally (e.g., configuration readers, logging services).
- **Scoped (`AddScoped`):**
  - A new instance of the service is created **once per client request (HTTP request)**.
  - Within the same HTTP request, all components requesting this service will receive the same instance.
  - Suitable for services that maintain state for the duration of a request (e.g., database contexts like Entity Framework Core's `DbContext`).
- **Transient (`AddTransient`):**
  - A new instance of the service is created **every time it is requested**.
  - Suitable for lightweight, stateless services that should not share state across different consumers or even within the same request (e.g., utility services, small calculation engines).

Example (Program.cs):
C#
```
var builder = WebApplication.CreateBuilder(args);

// Register services
builder.Services.AddSingleton<ISingletonService, SingletonService>();
builder.Services.AddScoped<IScopedService, ScopedService>();
builder.Services.AddTransient<ITransientService, TransientService>();
```

---

7. Filters

19. What are Filters in ASP.NET Core MVC and what are their types?

Filters allow you to execute logic before or after certain stages in the request processing pipeline. They help encapsulate cross-cutting concerns (e.g., logging, caching, authorization) without cluttering controller actions.

There are five main types of filters, executed in a specific order:

1. **Authorization Filters (`IAuthorizationFilter`)**: Run first, before anything else. Used for access control and authentication. If authorization fails, the pipeline is short-circuited.
2. **Resource Filters (`IResourceFilter`)**: Run after authorization. They can short-circuit the rest of the pipeline and are useful for caching or suppressing the rest of the pipeline.
3. **Action Filters (`IActionFilter` or `IAsyncActionFilter`)**: Run immediately before and after an action method is executed. Used for modifying action arguments, results, or logging.

4. **Exception Filters (`IExceptionFilter` or `IAsyncExceptionFilter`)**: Run only when an unhandled exception occurs during action execution or other filters. Used for global error handling and logging.
5. **Result Filters (`IResultFilter` or `IAsyncResultFilter`)**: Run immediately before and after an action result is executed. Useful for modifying the response before it's sent to the client (e.g., adding HTTP headers).

20. How do you create a custom Action Filter?

To create a custom action filter, you typically:

1. Create a class that inherits from `ActionFilterAttribute` (for synchronous logic) or implements `IAsyncActionFilter` (for asynchronous logic).
2. Override the `OnActionExecuting` (before action) and/or `OnActionExecuted` (after action) methods.

Example (synchronous logging filter):
C#
```
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.Extensions.Logging;

public class LogActionFilter : ActionFilterAttribute
{
    private readonly ILogger<LogActionFilter> _logger;

    public LogActionFilter(ILogger<LogActionFilter> logger)
    {
        _logger = logger;
    }

    public override void OnActionExecuting(ActionExecutingContext context)
    {
        _logger.LogInformation($"Executing action: 
{context.ActionDescriptor.DisplayName}");
        base.OnActionExecuting(context);
    }

    public override void OnActionExecuted(ActionExecutedContext context)
    {
        _logger.LogInformation($"Executed action: 
{context.ActionDescriptor.DisplayName}");
        base.OnActionExecuted(context);
    }
}
```

You can then apply this filter:

- Globally (in `Program.cs`): `builder.Services.AddControllersWithViews(options => options.Filters.Add<LogActionFilter>());`
- On a controller: `[LogActionFilter]`

- On an action method: `[LogActionFilter]`

---

8. Middleware

21. What is Middleware in ASP.NET Core?

Middleware components are software components that are assembled into an application pipeline to handle requests and responses. Each component can:

- Perform operations before or after the next component in the pipeline.
- Pass the request to the next middleware in the pipeline.
- Short-circuit the pipeline, meaning the request is not passed to subsequent middleware components.

Middleware is configured in `Program.cs` using methods like `app.Use...()`. The order in which middleware is added to the pipeline is crucial, as it determines the order of execution.

Common Middleware:

- `app.UseStaticFiles()`: Serves static files (HTML, CSS, JavaScript, images).
- `app.UseRouting()`: Matches incoming requests to endpoints.
- `app.UseAuthentication()`: Adds authentication support.
- `app.UseAuthorization()`: Adds authorization support.
- `app.UseEndpoints()`: Defines the routing endpoints (e.g., MVC controllers, Razor Pages).

Example (Program.cs):
C#
```
var builder = WebApplication.CreateBuilder(args);
// ... service registrations ...

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
else
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

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

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

app.Run();
```

22. What's the difference between `Use` and `Run` methods in the middleware pipeline?

- **`Use`**: Adds a middleware component to the pipeline that can perform logic both before and after subsequent middleware, and can also call the `next` delegate to pass the request to the next component. It's for components that participate in the full request/response cycle.
- **`Run`**: Adds a terminal middleware component to the pipeline. It short-circuits the pipeline and does not call `next`. Any middleware added after a `Run` middleware will never be executed. It's typically used for simple responses or error pages at the very end of the pipeline.

---

9. Configuration and Environment

23. What is `appsettings.json` and how do you read configuration from it?

`appsettings.json` is a JSON-based file used to store application configuration settings in ASP.NET Core. It supports hierarchical data and can be overridden by environment-specific files (e.g., `appsettings.Development.json`, `appsettings.Production.json`).

You read configuration using the `IConfiguration` interface, which is typically injected into your classes via Dependency Injection.

Example (appsettings.json):
JSON
```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "ConnectionStrings": {
    "DefaultConnection":
"Server=(localdb)\\mssqllocaldb;Database=MyDatabase;Trusted_Connection=Tru
e;MultipleActiveResultSets=true"
```

```
  },
  "MyCustomSetting": "Some Value"
}
```

Example (reading configuration in a controller):
C#
```
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Mvc;

public class HomeController : Controller
{
    private readonly IConfiguration _configuration;

    public HomeController(IConfiguration configuration)
    {
        _configuration = configuration;
    }

    public IActionResult Index()
    {
        string connectionString =
_configuration.GetConnectionString("DefaultConnection");
        string mySetting = _configuration["MyCustomSetting"];

        ViewBag.ConnectionString = connectionString;
        ViewBag.MySetting = mySetting;

        return View();
    }
}
```

24. How do you manage different environments (Development, Staging, Production) in ASP.NET Core?

ASP.NET Core uses the **`ASPNETCORE_ENVIRONMENT`** environment variable to determine the current environment. This allows for environment-specific configurations and behaviors.

- **Configuration Files:** You can have `appsettings.json`, `appsettings.Development.json`, `appsettings.Production.json`, etc. Settings in environment-specific files override those in `appsettings.json`.
- **Conditional Code:** You can use `app.Environment.IsDevelopment()`, `app.Environment.IsProduction()`, `app.Environment.IsStaging()`, or `app.Environment.IsEnvironment("CustomEnv")` in `Program.cs` (or `Startup.cs`) to configure middleware or services differently based on the environment.

Example (Program.cs):
C#
```
var builder = WebApplication.CreateBuilder(args);

// ... services ...
```

```
var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage(); // Detailed errors for development
}
else
{
    app.UseExceptionHandler("/Home/Error"); // Friendly error page for
production
    app.UseHsts();
}
// ... rest of middleware ...
```

---

10. Security and Authentication

25. How do you implement authentication and authorization in ASP.NET Core MVC?

- **Authentication:** Verifying the identity of a user. ASP.NET Core provides various authentication schemes (Cookie, JWT Bearer, OAuth, OpenID Connect). You configure them using `AddAuthentication` and `UseAuthentication` middleware.
- **Authorization:** Determining what an authenticated user is allowed to do. You use attributes like `[Authorize]` on controllers or actions.

Basic Setup (Cookie Authentication in Program.cs):

C#
```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddAuthentication(CookieAuthenticationDefaults.Authentica
tionScheme)
    .AddCookie(options =>
    {
        options.LoginPath = "/Account/Login";
        options.AccessDeniedPath = "/Account/AccessDenied";
    });

builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.
app.UseRouting();
app.UseAuthentication(); // Must be before UseAuthorization
app.UseAuthorization();

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

app.Run();
```

Controller/Action Authorization:
C#
```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

[Authorize] // Requires authentication for all actions in this controller
public class AdminController : Controller
{
    public IActionResult Index()
    {
        return View();
    }

    [Authorize(Roles = "Admin")] // Requires "Admin" role
    public IActionResult ManageUsers()
    {
        return View();
    }
}
```

26. What are Anti-Forgery Tokens and why are they important?

Anti-Forgery Tokens (also known as Request Verification Tokens) are a security mechanism used to prevent **Cross-Site Request Forgery (CSRF)** attacks. A CSRF attack tricks a user into unknowingly submitting a malicious request to a website where they are authenticated.

ASP.NET Core MVC generates a unique, cryptographically secure token and includes it in forms. When the form is submitted, the server validates this token. If the token is missing or invalid, the request is rejected.

You typically use `Html.AntiForgeryToken()` in your Razor forms:
HTML
```
<form asp-action="Create" method="post">
    @Html.AntiForgeryToken()
    <button type="submit">Submit</button>
</form>
```

And the `[ValidateAntiForgeryToken]` attribute on your controller action:
C#
```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Create(Product product)
{
    // ...
}
```

---

11. Performance and Scalability

27. What is Kestrel and why is it used in .NET Core?

Kestrel is a cross-platform web server for ASP.NET Core. It's the default and recommended server for ASP.NET Core applications.

- **High Performance:** Built from the ground up to be fast and lightweight.
- **Cross-Platform:** Runs on Windows, macOS, and Linux.
- **Self-contained:** Can run as a standalone process without requiring a separate web server like IIS or Nginx.
- **Reverse Proxy:** In production, Kestrel is often run behind a reverse proxy server (like IIS, Nginx, or Apache) for added security, load balancing, and static file serving capabilities.

28. How can you improve the performance of an ASP.NET Core MVC application?

- **Caching:**
  - **Output Caching:** Cache the entire response of an action method.
  - **In-Memory Caching:** Cache specific data in memory using `IMemoryCache`.
  - **Distributed Caching:** Use Redis or SQL Server to cache data across multiple servers.
- **Asynchronous Programming (`async`/`await`):** Use asynchronous actions to free up threads while waiting for I/O operations (database calls, external API calls) to complete, improving scalability.
- **Optimized Database Access:** Use efficient queries, proper indexing, and lazy loading carefully.
- **Client-Side Optimizations:** Minify and bundle JavaScript/CSS, lazy load images, use CDNs.
- **Response Compression:** Enable GZIP or Brotli compression for HTTP responses.
- **Middleware Optimization:** Order middleware efficiently, avoiding unnecessary processing.
- **HTTP/2:** Use HTTP/2 for multiplexing and header compression.
- **Logging:** Be mindful of excessive logging, especially in production.
- **Profiling:** Use profilers (e.g., Visual Studio Profiler, MiniProfiler) to identify performance bottlenecks.

---

12. Advanced Topics

29. What are Tag Helpers and how do they differ from HTML Helpers?

Tag Helpers are server-side components that participate in rendering HTML elements in Razor files. They enable server-side code to create and render HTML elements. They look like standard HTML tags but have `asp-` prefixes (e.g., `<form asp-action="Create">`).

Differences from HTML Helpers:

- **Syntax:** Tag Helpers use an HTML-like syntax, making Razor views more readable and closer to plain HTML. HTML Helpers use C# method calls (e.g., `@Html.TextBoxFor`).
- **IDE Support:** Tag Helpers offer better IntelliSense and design-time experience in Visual Studio.
- **Readability:** Generally considered more readable and maintainable for front-end developers familiar with HTML.
- **Composition:** Tag Helpers can be nested and compose effectively, similar to HTML elements.

Example (Tag Helper):
HTML
```html
<form asp-controller="Product" asp-action="Create" method="post">
    <label asp-for="Name"></label>
    <input asp-for="Name" />
    <span asp-validation-for="Name" class="text-danger"></span>
    <button type="submit">Create</button>
</form>
```

Equivalent HTML Helper:
HTML
```html
@using (Html.BeginForm("Create", "Product", FormMethod.Post))
{
    @Html.LabelFor(m => m.Name)
    @Html.TextBoxFor(m => m.Name)
    @Html.ValidationMessageFor(m => m.Name, "", new { @class = "text-
danger" })
    <button type="submit">Create</button>
}
```

30. What are View Components? When would you use them over Partial Views?

View Components are a new feature in ASP.NET Core MVC (replacing Child Actions from ASP.NET MVC 5) that are similar to partial views but are more powerful and self-contained. They are best suited for reusable UI elements with complex logic or database interaction.

Key Differences/Advantages over Partial Views:

- **Self-contained Logic:** View Components have their own C# class (logic) and Razor view (UI), making them truly self-contained. Partial views are just UI templates.
- **No Controller Dependencies:** View Components don't rely on controller actions. They are invoked directly from a view and handle their own data retrieval.
- **Testability:** Because they are self-contained, View Components are easier to unit test.
- **Asynchronous:** They fully support asynchronous operations.
- **Model Binding:** They don't participate in model binding directly; they receive parameters.

When to use View Components:

- Side-bar navigation menus.

- Shopping cart summaries.
- Login forms.
- Tag clouds.
- Recently published articles lists.

When to use Partial Views:

- Simple, reusable UI snippets with no complex logic.
- Rendering a subset of a model within a parent view.

Combining ASP.NET Core MVC with Entity Framework Core (EF Core) is a very common and powerful way to build web applications. Here are interview questions and answers specifically focusing on EF Core within an MVC context.

---

1. Entity Framework Core Fundamentals

1. What is Entity Framework Core (EF Core) and what is its purpose?

Entity Framework Core (EF Core) is a lightweight, cross-platform, and open-source **Object-Relational Mapper (ORM)** for .NET. Its primary purpose is to enable .NET developers to work with a database using **.NET objects (C# classes)** instead of writing raw SQL queries.

It simplifies data access by:

- **Mapping objects to database tables:** You define your data model using C# classes (entities), and EF Core handles the translation between these objects and the relational database schema.
- **Automating CRUD operations:** It automatically generates and executes SQL commands for **Create, Read, Update, and Delete (CRUD)** operations based on your object interactions.
- **Reducing boilerplate code:** It significantly reduces the amount of data access code you need to write and maintain.
- **Providing LINQ support:** Allows you to query your database using **Language Integrated Query (LINQ)**, which is strongly typed and checked at compile time.

2. Why use an ORM like EF Core instead of raw ADO.NET or Dapper?

- **Increased Productivity:** Developers can focus on domain logic and objects rather than writing repetitive SQL queries and mapping data manually. This speeds up development.
- **Type Safety:** LINQ queries are strongly typed, catching errors at compile time rather than runtime, which is a significant advantage over string-based SQL queries.
- **Database Agnostic:** EF Core supports various database providers (SQL Server, PostgreSQL, MySQL, SQLite, etc.), allowing you to change the underlying database with minimal code changes.
- **Abstraction:** It abstracts away the complexities of database interaction, connection management, and SQL query generation.
- **Features:** Provides built-in features like change tracking, migrations, lazy/eager/explicit loading, and concurrency handling.

While EF Core is powerful, for highly optimized, complex queries or scenarios where maximum performance is critical and full control over SQL is desired, lower-level ORMs like Dapper or raw ADO.NET might be considered.

3. Explain the role of `DbContext` in EF Core.

The **`DbContext`** class is the primary class in EF Core responsible for interacting with the database. It represents a **session with the database** and acts as a combination of the **Unit of Work** and **Repository** patterns.

Key responsibilities of `DbContext`:

- **Database Connection:** Manages the connection to the database.
- **Querying Data:** Provides `DbSet<TEntity>` properties that represent collections of your entities (tables) from which you can execute LINQ queries.
- **Change Tracking:** Tracks changes made to entities loaded from the database. When `SaveChanges()` is called, it detects these changes and generates the appropriate SQL `INSERT`, `UPDATE`, or `DELETE` commands.
- **Object Materialization:** Converts the data retrieved from the database into .NET objects.
- **Transaction Management:** Provides basic transaction capabilities (though explicit transaction control is also possible).

You typically create a class that inherits from `DbContext` and add `DbSet<TEntity>` properties for each entity you want to expose.

Example:
C#
```
public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options) : base(options)
    {
    }

    public DbSet<Product> Products { get; set; }
    public DbSet<Category> Categories { get; set; }

    // Optional: Further configuration using Fluent API
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        // Example: Configure Product entity
        modelBuilder.Entity<Product>()
            .Property(p => p.Name)
            .IsRequired()
            .HasMaxLength(200);

        base.OnModelCreating(modelBuilder);
    }
}
```

---

2. Data Modeling and Schema Management

4. Differentiate between Code-First and Database-First approaches in EF Core.

These are two primary approaches for generating and managing your EF Core model and database schema:

- **Code-First (Recommended for new projects):**
  - You start by defining your **data model using C# classes (POCOs - Plain Old CLR Objects)**.
  - EF Core then uses these classes to **generate the database schema** (tables, columns, relationships) for you, typically through **Migrations**.
  - **Pros:** Favors a domain-driven design, provides full control over the data model in code, works well with agile development, and makes versioning schema changes easy with migrations.
  - **Cons:** Less suitable for existing, complex databases where manual database design is critical.
- **Database-First (Suitable for existing databases):**
  - You start with an **existing database schema**.
  - EF Core **reverse-engineers** the database into C# entity classes and a `DbContext` class.
  - **Pros:** Ideal for working with legacy databases, ensures the C# model accurately reflects the existing schema, and can be easier if DBAs strictly control database design.
  - **Cons:** Changes to the database require re-scaffolding the model, which can overwrite manual code changes. Less control over the C# model generated.

5. What are Migrations in EF Core and why are they important?

Migrations are an EF Core feature that allows you to **evolve your database schema over time** as your application's entity model changes, without losing existing data.

Importance:

- **Schema Evolution:** They track changes to your entity classes (e.g., adding a new property, changing a data type) and generate the necessary SQL scripts to update the database.
- **Version Control for Database:** Migrations provide a versioned history of your schema changes, similar to how source control manages code changes.
- **Automated Updates:** You can apply migrations to update your database to a specific version, ensuring consistency across different development, staging, and production environments.
- **Collaboration:** Facilitates team development by providing a structured way to manage database changes introduced by multiple developers.

Typical Workflow:

1. Modify your entity classes (e.g., add a `Description` property to `Product`).
2. Run `dotnet ef migrations add <MigrationName>` (e.g., `AddProductDescription`). This generates a migration file with `Up()` (to apply changes) and `Down()` (to revert changes) methods.
3. Run `dotnet ef database update`. This executes the `Up()` method of pending migrations to update the database.

6. How do you configure relationships (one-to-many, many-to-many) in EF Core?

EF Core uses conventions, Data Annotations, and the Fluent API to configure relationships:

- **Conventions:** EF Core often infers relationships based on naming conventions (e.g., `CategoryId` property in `Product` class will be recognized as a foreign key to `Category`).
- **Data Annotations:** Attributes applied to properties in your entity classes.
  - `[ForeignKey("PropertyName")]`
  - `[InverseProperty("NavigationPropertyName")]`
- **Fluent API (Recommended for complex configurations):** More powerful and flexible, configured by overriding the `OnModelCreating` method in your `DbContext`.

  Example (One-to-Many: Category has many Products):
  C#
```
// Category.cs
public class Category
{
    public int CategoryId { get; set; }
    public string Name { get; set; }
    public ICollection<Product> Products { get; set; } // Navigation
property
}

// Product.cs
public class Product
{
    public int ProductId { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }

    public int CategoryId { get; set; } // Foreign Key property
    public Category Category { get; set; } // Navigation property
}

// ApplicationDbContext.cs (Fluent API)
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Product>()
        .HasOne(p => p.Category)           // Product has one Category
        .WithMany(c => c.Products)         // Category has many Products
        .HasForeignKey(p => p.CategoryId); // Foreign key is CategoryId

    base.OnModelCreating(modelBuilder);
}
```

---

3. Querying Data

7. Explain LINQ to Entities and how it works.

LINQ (Language Integrated Query) to Entities is the component of EF Core that allows you to write queries against your `DbContext` using C# (or VB.NET) query syntax or method syntax. EF Core then translates these LINQ queries into **SQL queries** that are executed against the underlying database.

How it works:

1. You write a LINQ query against a `DbSet<TEntity>`.
2. EF Core's query provider analyzes the LINQ expression tree.
3. It translates the expression tree into an equivalent SQL query.
4. The SQL query is executed against the database.
5. The results (data rows) are materialized back into .NET objects.

Benefits:

- **Strongly Typed:** Compiler checks for errors, improving reliability.
- **IntelliSense:** Provides auto-completion and type hints in IDEs.
- **Readability:** Queries are often more readable than raw SQL, especially for complex operations.
- **Abstraction:** You don't need to know the exact SQL syntax for different database systems.

Example:

C#
```
// Method syntax
var activeProducts = _context.Products
    .Where(p => p.IsActive && p.Price > 10)
    .OrderBy(p => p.Name)
    .ToList(); // ToList() executes the query

// Query syntax
var expensiveProducts = from p in _context.Products
                        where p.Price > 50
                        orderby p.Name ascending
                        select p;
List<Product> productsList = expensiveProducts.ToList();
```

8. Differentiate between Lazy Loading and Eager Loading in EF Core.

These are strategies for loading related entities (navigation properties):

- **Lazy Loading:**
  - Related entities are **not loaded from the database until they are explicitly accessed**.
  - Requires navigation properties to be `virtual`.
  - **Pros:** Reduces initial data retrieval, loads only what's needed.
  - **Cons:** Can lead to the **"N+1 query problem"** where accessing a collection of parent entities and then looping through them to access their related children results in N+1 database roundtrips (N for children, 1 for parents), significantly impacting performance.
  - **Enable:** Install `Microsoft.EntityFrameworkCore.Proxies` and call `optionsBuilder.UseLazyLoadingProxies()` in `DbContext.OnConfiguring` or `Program.cs`.

C#
```
// Example with lazy loading enabled:
var products = _context.Products.ToList(); // Loads products only
```

```
foreach (var product in products)
{
    // Category is loaded *here* for each product, causing N queries
    Console.WriteLine($"{product.Name} - {product.Category.Name}");
}
```

- **Eager Loading:**
  - Related entities are **loaded from the database along with the main entity in a single query**.
  - Achieved using the `.Include()` method in your LINQ query.
  - **Pros:** Avoids the N+1 query problem by fetching all necessary data in one roundtrip, generally better performance for scenarios where related data is always needed.
  - **Cons:** Can fetch more data than necessary if not all related data is always used, potentially increasing memory consumption.

C#
```
// Example with eager loading:
var productsWithCategories = _context.Products
    .Include(p => p.Category) // Eagerly loads the Category
    .ToList();
// All categories are already loaded, no extra queries
foreach (var product in productsWithCategories)
{
    Console.WriteLine($"{product.Name} - {product.Category.Name}");
}
```

9. When would you use `AsNoTracking()`?

The `.AsNoTracking()` extension method tells EF Core **not to track the entities** returned by a query. This means that:

- EF Core will **not perform change tracking** on these entities.
- Changes made to these entities will **not be persisted** to the database when `SaveChanges()` is called.
- No snapshots of the entities are taken, reducing memory overhead.

When to use it:

- **Read-Only Scenarios:** For queries where you only need to display data and don't intend to modify or update it (e.g., displaying a list of products on a public page).
- **Performance Optimization:** Improves query performance, especially for large datasets, as EF Core doesn't need to set up change tracking.
- **Detached Entities:** If you plan to load an entity, detach it, and then attach a modified version later.

  C#
  ```
  var products = _context.Products.AsNoTracking().ToList();
  // 'products' are not tracked, changes won't be saved unless re-attached.
  ```

---

4. Saving Data and Concurrency

10. How does EF Core track changes and what are the entity states?

EF Core's `DbContext` includes a **Change Tracker** that monitors the state of entities loaded into the context. When `SaveChanges()` is called, the Change Tracker detects differences between the current state of the entities and their original state (snapshot taken when loaded or added), then generates and executes the appropriate SQL commands.

An entity can be in one of five states:

- **`Added`**: The entity is new and has not yet been saved to the database. It will be inserted.
  - `_context.Products.Add(newProduct);`
- **`Modified`**: An existing entity whose property values have been changed. It will be updated.
  - `_context.Entry(existingProduct).State = EntityState.Modified;` or just modify properties after loading.
- **`Deleted`**: An existing entity marked for deletion from the database.
  - `_context.Products.Remove(productToDelete);`
- **`Unchanged`**: The entity has been loaded from the database, and no changes have been detected.
- **`Detached`**: The entity is not currently being tracked by the `DbContext`. This happens for newly created entities before `Add()` is called, or if an entity is explicitly detached.

11. How do you handle concurrency conflicts in EF Core?

Concurrency conflicts occur when multiple users (or processes) attempt to update the same data simultaneously, and one user's changes overwrite another's without either user being aware. EF Core primarily supports **optimistic concurrency control**.

Optimistic Concurrency Steps:

1. **Detection:** You configure a property in your entity as a **concurrency token**. This is usually a `rowversion` (SQL Server `timestamp`) column in the database, or a `[ConcurrencyCheck]` attribute on a property (e.g., a `ModifiedDate` column).
   - **`rowversion` (recommended for SQL Server):** A `byte[]` property marked with `[Timestamp]` or `IsRowVersion()` in Fluent API. The database automatically updates this column on every row modification.
   - **`[ConcurrencyCheck]`:** For any other property you want to use for concurrency checking.
2. **Conflict Occurs:** When `SaveChanges()` is called, EF Core includes the original value of the concurrency token in the `WHERE` clause of the `UPDATE` or `DELETE` statement. If the record in the database has a different token value (meaning it was changed by someone else), no rows will be affected.
3. **Exception:** EF Core detects that zero rows were affected and throws a **`DbUpdateConcurrencyException`**.
4. **Resolution:** In your code, you catch this exception and implement a resolution strategy:
   - **Client Wins:** Overwrite the database with the current client's changes.

- **Database Wins:** Refresh the client's entity with the latest values from the database, discarding client changes.
- **Merge/Custom Logic:** Present both versions to the user and let them decide how to merge the changes.

Example (using `rowversion`):
C#
```
public class Product
{
    public int ProductId { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    [Timestamp] // Data annotation for concurrency token
    public byte[] RowVersion { get; set; }
}

// In your controller/service:
public async Task<IActionResult> UpdateProduct(Product product)
{
    _context.Entry(product).State = EntityState.Modified;
    try
    {
        await _context.SaveChangesAsync();
        return Ok(product);
    }
    catch (DbUpdateConcurrencyException ex)
    {
        // Get the conflicting entity
        var entry = ex.Entries.Single();
        var databaseValues = entry.GetDatabaseValues();
        var databaseProduct = (Product)databaseValues.ToObject();

        // Example: Database wins strategy (refresh client's entity)
        // entry.OriginalValues.SetValues(databaseValues);
        // return Conflict(new { Message = "Record was modified by another
user. Your changes were not saved.", DatabaseProduct = databaseProduct });

        // Example: Client wins strategy (overwrite database values)
        entry.OriginalValues.SetValues(entry.CurrentValues);
        await _context.SaveChangesAsync(); // Retry saving
        return Ok(new { Message = "Your changes overwritten previous
changes.", Product = product });
    }
}
```

---

5. Performance and Asynchronous Operations

12. How do you perform asynchronous operations with EF Core? Why is it important in ASP.NET Core MVC?

EF Core provides **asynchronous versions** of its methods (e.g., `ToListAsync()`, `FirstOrDefaultAsync()`, `SaveChangesAsync()`, `AddAsync()`). You use the `async` and `await` keywords to implement them.

Why it's important in ASP.NET Core MVC:

- **Scalability:** In web applications, I/O-bound operations (like database calls) can block the current thread while waiting for the operation to complete. Using `async`/`await` frees up the thread to handle other incoming requests, improving the application's overall throughput and scalability.
- **Responsiveness:** Prevents the server from becoming unresponsive under heavy load.
- **Resource Utilization:** Efficiently utilizes server resources by not tying up threads unnecessarily.

Example:
C#
```csharp
public class ProductsController : Controller
{
    private readonly ApplicationDbContext _context;

    public ProductsController(ApplicationDbContext context)
    {
        _context = context;
    }

    // Synchronous (less scalable for I/O bound operations)
    public IActionResult IndexSync()
    {
        var products = _context.Products.ToList();
        return View(products);
    }

    // Asynchronous (recommended for I/O bound operations in web apps)
    public async Task<IActionResult> Index()
    {
        // The thread is released while waiting for the database call
        var products = await _context.Products.ToListAsync();
        return View(products);
    }

    [HttpPost]
    public async Task<IActionResult> Create(Product product)
    {
        if (ModelState.IsValid)
        {
            await _context.Products.AddAsync(product); // Asynchronous add
            await _context.SaveChangesAsync(); // Asynchronous save
            return RedirectToAction(nameof(Index));
        }
        return View(product);
    }
}
```

13. What are some common performance pitfalls with EF Core and how can you mitigate them?

- **N+1 Query Problem (Lazy Loading):**
  - **Mitigation:** Use **Eager Loading (`.Include()`)** or **Explicit Loading** for related data that is always needed.
- **Loading Too Much Data (`SELECT *` instead of specific columns):**
  - **Mitigation:** Use **Projection (`.Select()`)** to retrieve only the necessary columns.
  - `var productNames = await _context.Products.Select(p => p.Name).ToListAsync();`
- **Excessive Roundtrips/Small Batches:**
  - **Mitigation:** Batch operations where possible (e.g., `AddRange()` for multiple inserts).
  - Consider third-party libraries for bulk operations (e.g., `Z.EntityFramework.Plus.EFCore` for bulk inserts/updates/deletes).
- **Not Using `AsNoTracking()` for Read-Only Queries:**
  - **Mitigation:** Apply `AsNoTracking()` to queries that don't require change tracking.
- **Inefficient Queries (e.g., client-side evaluation):**
  - **Mitigation:** Ensure complex LINQ queries are fully translated to SQL. Methods like `.ToList()` or `.AsEnumerable()` can force client-side evaluation if placed too early, leading to fetching more data than needed.
  - Use `.Where()`, `.OrderBy()`, `.GroupBy()` before materializing to force server-side execution.
- **Misconfigured Caching:**
  - **Mitigation:** Implement caching strategies (in-memory, distributed) for frequently accessed, slowly changing data.
- **Poor Database Indexing:**
  - **Mitigation:** Ensure your database has appropriate indexes on columns used in `WHERE`, `ORDER BY`, and `JOIN` clauses.
- **Improper `DbContext` Lifetime:**
  - **Mitigation:** `DbContext` should typically have a **scoped** lifetime in web applications (one instance per request). Avoid singletons.

---

6. Advanced EF Core Concepts

14. Explain what `DbSet<TEntity>` is.

A **`DbSet<TEntity>`** represents a collection of all entities in the context, or that can be queried from the database, of a given type. It's essentially a conceptual representation of a database table within your `DbContext`.

- It acts as the primary entry point for **LINQ queries** against a specific entity type.
- It allows you to perform **CRUD operations** (`Add`, `Remove`, `Update`, `Find`, `Attach`).
- It exposes methods that translate into database operations.

Example:

public DbSet<Product> Products { get; set; }

15. How do you implement custom validation in EF Core beyond Data Annotations?

While Data Annotations are great for simple validation rules, for more complex, cross-property, or business logic validation, you can use:

- **`IValidatableObject` Interface:** Implement this interface on your entity class. The `Validate` method allows you to return a collection of `ValidationResult` objects. This validation runs during `SaveChanges()`.

  C#
  ```
  public class Product : IValidatableObject
  {
      public int ProductId { get; set; }
      public string Name { get; set; }
      public decimal Price { get; set; }
      public DateTime AvailableFrom { get; set; }

      public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
      {
          if (Price <= 0)
          {
              yield return new ValidationResult("Price must be greater than zero.", new[] { nameof(Price) });
          }
          if (AvailableFrom > DateTime.Now)
          {
              yield return new ValidationResult("Product cannot be available in the future.", new[] { nameof(AvailableFrom) });
          }
      }
  }
  ```

- **Fluent API `HasCheckConstraint`:** For database-level check constraints.
- **Service Layer Validation:** Implement validation logic in a service layer before passing data to the `DbContext`. This is often the most flexible approach for complex business rules.
- **FluentValidation Library:** A popular third-party library that provides a fluent interface for building strong-typed validation rules, separate from your entity classes.

You're looking for SQL Server/MySQL interview questions and answers, focusing on general database concepts, SQL queries, and specific features relevant to these relational database management systems. Let's dive in!

---

1. Relational Database Concepts

1. What is a Relational Database Management System (RDBMS)?

A **Relational Database Management System (RDBMS)** is a software system used to create, manage, and query relational databases. It stores data in tables (relations), which are organized into rows (records/tuples) and columns (fields/attributes). Data is related across tables using common columns, enabling powerful data retrieval and integrity. Both SQL Server and MySQL are prominent examples of RDBMS.

2. What are ACID properties in the context of database transactions?

ACID is an acronym that defines a set of properties guaranteeing that database transactions are processed reliably.

- **Atomicity:** A transaction is treated as a single, indivisible unit of work. Either all of its operations are completed successfully, or none of them are. If any part fails, the entire transaction is rolled back.
  - *Analogy:* Transferring money between bank accounts. Either the debit from one account and credit to another both happen, or neither does.
- **Consistency:** A transaction brings the database from one valid state to another. It ensures that data integrity rules (like primary key constraints, foreign key constraints, and check constraints) are maintained before and after the transaction.
- **Isolation:** Concurrent transactions are executed in such a way that they appear to be executed sequentially. Each transaction is unaware of other concurrent transactions, preventing interference and ensuring data integrity.
- **Durability:** Once a transaction has been committed, its changes are permanent and will survive system failures (e.g., power outages, crashes). Committed data is written to non-volatile storage.

3. Explain the difference between a Primary Key and a Foreign Key.

- **Primary Key (PK):** A column or set of columns in a table that uniquely identifies each row in that table.
  - **Rules:** Must contain unique values, cannot contain NULL values (NOT NULL constraint). A table can have only one Primary Key.
  - **Purpose:** Ensures entity integrity, provides a default clustered index (in SQL Server by default), and serves as the target for Foreign Key relationships.
  - *Example:* `CustomerID` in a `Customers` table.
- **Foreign Key (FK):** A column or set of columns in one table that refers to the Primary Key (or a Candidate Key) in another table (or the same table).

  - **Rules:** Its values must match values in the referenced Primary Key column, or be NULL.
  - **Purpose:** Establishes and enforces a link between two tables, maintaining referential integrity.
  - *Example:* `CustomerID` in an `Orders` table, referencing `CustomerID` in the `Customers` table.

4. What is Normalization in database design? What are its benefits?

Normalization is a systematic process of structuring a relational database to reduce data redundancy and improve data integrity. It involves organizing columns and tables to ensure that their dependencies are correctly enforced. The process involves breaking down larger tables into smaller, related tables and defining relationships between them.

Benefits:

- **Reduced Data Redundancy:** Avoids storing the same piece of data multiple times, saving storage space.
- **Improved Data Integrity:** Reduces the chances of inconsistent data. Changes only need to be made in one place.
- **Easier Maintenance:** Updates, insertions, and deletions become simpler and less prone to errors.
- **Faster Query Performance:** Smaller, more focused tables can sometimes lead to faster query execution, especially for update operations.

Normal Forms (Common ones):

- **1st Normal Form (1NF):** Eliminate repeating groups and ensure all columns contain atomic (single) values.
- **2nd Normal Form (2NF):** Be in 1NF and ensure all non-key attributes are fully dependent on the entire primary key.
- **3rd Normal Form (3NF):** Be in 2NF and ensure all non-key attributes are not dependent on other non-key attributes (eliminate transitive dependencies).
- **Boyce-Codd Normal Form (BCNF):** A stricter version of 3NF, ensuring every determinant is a candidate key.

5. What is Denormalization and when might you use it?

Denormalization is the process of intentionally introducing redundancy into a database by combining data from multiple tables into one, or by adding redundant columns to a table. This is typically done after normalization, often to improve read performance.

When to use it:

- **Read-Heavy Applications:** When reporting or analytical queries frequently join many tables, denormalization can significantly speed up data retrieval by reducing the number of joins.
- **Performance Bottlenecks:** If complex joins or frequent aggregation queries are causing performance issues.

- **Summary Tables:** Creating summary or aggregate tables (e.g., daily sales totals) that are pre-calculated to avoid real-time complex computations.
- **Data Warehousing:** Often employed in data warehouses where query performance for reporting is paramount, and data integrity during updates is less frequent.

  Trade-offs: Increases data redundancy, which can lead to data inconsistencies and makes data modification more complex (requires more careful handling of updates).

---

2. SQL Queries

6. Write a SQL query to select all employees who earn more than the average salary of their department.

Let's assume an `Employees` table with `EmployeeID`, `Name`, `Salary`, and `DepartmentID` columns.
SQL

```sql
SELECT
    e.EmployeeID,
    e.Name,
    e.Salary,
    e.DepartmentID
FROM
    Employees e
JOIN (
    SELECT
        DepartmentID,
        AVG(Salary) AS AvgDepartmentSalary
    FROM
        Employees
    GROUP BY
        DepartmentID
) AS DeptAvg ON e.DepartmentID = DeptAvg.DepartmentID
WHERE
    e.Salary > DeptAvg.AvgDepartmentSalary;
```

Explanation:

1. **Inner Subquery (`DeptAvg`):** Calculates the `AVG(Salary)` for each `DepartmentID`.
2. **Outer Query:** Joins the `Employees` table with the result of the subquery on `DepartmentID`.
3. **`WHERE` Clause:** Filters employees whose `Salary` is greater than their `DepartmentID`'s `AvgDepartmentSalary`.

7. Explain `JOIN` types (INNER, LEFT, RIGHT, FULL) with examples.

`JOIN` clauses combine rows from two or more tables based on a related column between them.

- **`INNER JOIN`**: Returns only the rows that have matching values in **both** tables.

SQL
```sql
-- Get employees and their matching departments
SELECT E.Name, D.DepartmentName
FROM Employees E
INNER JOIN Departments D ON E.DepartmentID = D.DepartmentID;
```

- **`LEFT JOIN` (or `LEFT OUTER JOIN`)**: Returns all rows from the **left** table, and the matching rows from the right table. If there is no match in the right table, `NULL` values are returned for columns from the right table.

  SQL
  ```sql
  -- Get all employees, and their department if they have one
  SELECT E.Name, D.DepartmentName
  FROM Employees E
  LEFT JOIN Departments D ON E.DepartmentID = D.DepartmentID;
  ```

- **`RIGHT JOIN` (or `RIGHT OUTER JOIN`)**: Returns all rows from the **right** table, and the matching rows from the left table. If there is no match in the left table, `NULL` values are returned for columns from the left table.

  SQL
  ```sql
  -- Get all departments, and any employees in them
  SELECT E.Name, D.DepartmentName
  FROM Employees E
  RIGHT JOIN Departments D ON E.DepartmentID = D.DepartmentID;
  ```

- **`FULL JOIN` (or `FULL OUTER JOIN`)**: Returns all rows when there is a match in either the left or the right table. If no match, `NULL` values are returned for columns from the non-matching side.
  - *Note: MySQL does not directly support `FULL OUTER JOIN`. It can be simulated using `UNION` of `LEFT JOIN` and `RIGHT JOIN`.*

  SQL
  ```sql
  -- SQL Server/PostgreSQL example
  -- Get all employees and all departments, regardless of match
  SELECT E.Name, D.DepartmentName
  FROM Employees E
  FULL OUTER JOIN Departments D ON E.DepartmentID = D.DepartmentID;

  -- MySQL simulation:
  SELECT E.Name, D.DepartmentName
  FROM Employees E
  LEFT JOIN Departments D ON E.DepartmentID = D.DepartmentID
  UNION
  SELECT E.Name, D.DepartmentName
  FROM Employees E
  RIGHT JOIN Departments D ON E.DepartmentID = D.DepartmentID
  WHERE E.DepartmentID IS NULL; -- Only include unmatched right rows from
  the right join to avoid duplicates from INNER JOIN matches
  ```

8. What is the difference between `WHERE` and `HAVING` clauses?

- **`WHERE` clause:**
  - Filters rows **before** they are grouped.
  - Applies to individual rows.
  - Cannot use aggregate functions (like `SUM()`, `AVG()`, `COUNT()`) directly in its condition.
- **`HAVING` clause:**
  - Filters groups of rows **after** they have been formed by the `GROUP BY` clause.
  - Applies to aggregated results (groups).
  - Can use aggregate functions in its condition.

Example:
SQL
```sql
-- Select departments where the average salary is greater than 60000
SELECT
    DepartmentID,
    AVG(Salary) AS AverageSalary
FROM
    Employees
GROUP BY
    DepartmentID
HAVING
    AVG(Salary) > 60000;

-- Select employees in departments where the average salary is greater
than 60000
SELECT
    EmployeeID,
    Name,
    DepartmentID,
    Salary
FROM
    Employees
WHERE
    DepartmentID IN (SELECT DepartmentID FROM Employees GROUP BY
DepartmentID HAVING AVG(Salary) > 60000);
```

9. What are Window Functions (Analytic Functions) and provide an example.

Window Functions perform a calculation across a set of table rows that are related to the current row. Unlike aggregate functions (which collapse rows into a single summary row), window functions return a value for each row, operating over a "window" of rows defined by the `OVER` clause.

Common uses: Ranking, moving averages, cumulative sums, lead/lag values.

Syntax: `function_name (expression) OVER ([partition_by_clause] [order_by_clause])`

Example (Ranking employees by salary within each department):
SQL

```
SELECT
    EmployeeID,
    Name,
    DepartmentID,
    Salary,
    ROW_NUMBER() OVER (PARTITION BY DepartmentID ORDER BY Salary DESC) AS
DepartmentRank
FROM
    Employees;
```

Explanation:

- `ROW_NUMBER()`: Assigns a unique, sequential number to each row within its partition.
- `PARTITION BY DepartmentID`: Divides the result set into partitions (groups) based on `DepartmentID`. The ranking restarts for each new department.
- `ORDER BY Salary DESC`: Orders the rows within each partition by `Salary` in descending order.

Other common window functions: `RANK()`, `DENSE_RANK()`, `NTILE()`, `LAG()`, `LEAD()`, `SUM() OVER()`, `AVG() OVER()`.

10. Differentiate between `DELETE`, `TRUNCATE`, and `DROP` commands.

These commands are used to remove data or database objects, but they operate differently:

- **`DELETE`**:
  - **Purpose:** Removes specific rows from a table based on a `WHERE` clause. If no `WHERE` clause, it removes all rows.
  - **Behavior:** It's a DML (Data Manipulation Language) command. It logs each deleted row, allowing for rollback. Triggers can be fired.
  - **Performance:** Slower for large tables as it involves logging and potentially firing triggers.
  - **Auto-increment Reset:** Does not reset the auto-increment counter.
  - **Syntax:** `DELETE FROM TableName WHERE Condition;`
- **`TRUNCATE TABLE`**:
  - **Purpose:** Removes all rows from a table, effectively emptying it.
  - **Behavior:** It's a DDL (Data Definition Language) command. It deallocates the data pages used by the table, making it very fast. It's minimally logged and cannot be rolled back easily in a live transaction (though implicitly rolled back if the surrounding transaction fails). Triggers are generally *not* fired.
  - **Performance:** Much faster than `DELETE` for large tables because it doesn't log individual row deletions.
  - **Auto-increment Reset:** Resets the auto-increment counter to its seed value (usually 0 or 1).
  - **Syntax:** `TRUNCATE TABLE TableName;`
- **`DROP TABLE`**:
  - **Purpose:** Removes the entire table structure (schema and data) from the database.
  - **Behavior:** It's a DDL command. It irrevocably deletes the table and all its associated objects (indexes, constraints, triggers). Cannot be rolled back.
  - **Performance:** Very fast, as it just deletes metadata and deallocates storage.

- **Auto-increment Reset:** Not applicable, as the table itself is gone.
- **Syntax:** `DROP TABLE TableName;`

Feature	DELETE	TRUNCATE TABLE	DROP TABLE
Rows Removed	Selected or All	All rows	All rows & table
Logging	Fully logged	Minimally logged	Logged
Rollback	Possible	Not easily (implicit)	Not possible
Triggers Fired	Yes	No	No
Auto-increment	No reset	Resets	Not applicable
Command Type	DML	DDL	DDL

---

3. Indexing and Performance

11. What is an Index in a database and why is it used?

An **Index** is a data structure (like a B-tree) that improves the speed of data retrieval operations on a database table. It provides a quick lookup path to data, similar to an index in a book.

Why it's used:

- **Faster Data Retrieval:** Significantly speeds up `SELECT` queries, especially for `WHERE` clauses, `ORDER BY` clauses, and `JOIN` operations.
- **Unique Constraints:** Can enforce uniqueness on a column (e.g., a unique index or primary key).
- **Performance for Joins:** Improves the performance of joins by allowing the database to quickly find matching rows in related tables.

Trade-offs:

- **Increased Storage:** Indexes consume disk space.
- **Slower Writes:** `INSERT`, `UPDATE`, and `DELETE` operations become slower because the index also needs to be updated.

12. Differentiate between Clustered and Non-Clustered Indexes (SQL Server).

This distinction is particularly important in SQL Server:

- **Clustered Index:**

    - **Data Storage:** Determines the **physical order** of data rows in the table. The table's data rows are stored in the same order as the clustered index.
    - **Number per table:** Only **one** clustered index per table (since data can only be sorted physically in one way).
    - **Leaf Level:** The leaf level of a clustered index *is* the data pages of the table.
    - **Primary Key:** By default, the Primary Key constraint creates a clustered index on the table in SQL Server (unless specified otherwise).
    - **Benefit:** Excellent for range queries (`BETWEEN`, `>`, `<`), and queries that return large sets of ordered data.
- **Non-Clustered Index:**
    - **Data Storage:** Does **not** determine the physical order of data rows. It's a separate structure that contains the indexed key values and pointers (row locators or clustered index keys) to the actual data rows.
    - **Number per table:** A table can have **multiple** non-clustered indexes.
    - **Leaf Level:** The leaf level contains the index keys and pointers to the data rows.
    - **Benefit:** Good for specific lookups, and can cover queries (if all requested columns are in the index).
- *Note: In MySQL (specifically InnoDB storage engine), primary keys are always clustered. Secondary (non-clustered) indexes contain the primary key value as a pointer to the data.*

13. What is an Execution Plan and how is it useful?

An **Execution Plan** (also known as a Query Plan) is a graphical or textual representation of the steps that the database query optimizer will perform to execute a SQL query. It shows how the database engine intends to access, join, filter, and sort the data.

How it's useful:

- **Performance Tuning:** It's an essential tool for identifying performance bottlenecks in slow queries.
- **Index Analysis:** Helps determine if existing indexes are being used effectively or if new indexes are needed.
- **Join Optimization:** Shows the order in which tables are joined and the join algorithms used.
- **Resource Consumption:** Provides estimates of I/O, CPU, and memory usage for different query operations.
- **Troubleshooting:** Helps understand why a query isn't performing as expected.

You can typically view execution plans in SQL Server Management Studio (SSMS) using "Display Actual Execution Plan" or "Display Estimated Execution Plan." Similar tools exist for MySQL (e.g., `EXPLAIN` command).

---

4. Stored Procedures, Functions, and Triggers

14. What is a Stored Procedure and what are its advantages?

A **Stored Procedure** is a pre-compiled collection of one or more SQL statements (and control-of-flow statements like `IF`, `WHILE`) stored in the database. It can accept input parameters and return output parameters or result sets.

Advantages:

- **Performance:** Pre-compiled nature can lead to faster execution compared to sending raw SQL queries repeatedly.
- **Reduced Network Traffic:** Instead of sending multiple SQL statements, only the procedure call is sent.
- **Security:** Users can be granted permissions to execute procedures without having direct access to the underlying tables. This reduces the attack surface.
- **Reusability:** Code can be written once and called from multiple applications or parts of the same application.
- **Data Integrity:** Can enforce complex business rules and data consistency that might be difficult with simple constraints.
- **Centralized Logic:** Business logic can be encapsulated within the database, ensuring consistent application across different clients.

15. Differentiate between a Stored Procedure and a User-Defined Function (UDF).

Feature	Stored Procedure	User-Defined Function (UDF)
Purpose	Perform a task, modify data	Compute a scalar value or table
Data Modification	Can perform DML (INSERT, UPDATE, DELETE)	Cannot perform DML (read-only)
Return Value	Can return 0, 1, or multiple result sets; can use `OUTPUT` parameters.	Must return a single scalar value or a table.
Calling Method	Executed using `EXEC` or `CALL` statement.	Can be used in `SELECT`, `WHERE`, `HAVING` clauses, or as an expression.
Side Effects	Can have side effects (modify data, send emails).	Should not have side effects (pure functions are preferred).
Error Handling	Supports `TRY...CATCH` blocks.	Limited error handling.
Transactions	Can manage transactions explicitly.	Cannot manage transactions.

Feature	Stored Procedure	User-Defined Function (UDF)
Usage	Business logic, complex operations.	Calculations, data transformations.

16. What is a Trigger and when would you use one?

A **Trigger** is a special type of stored procedure that automatically executes (fires) when a specific data modification event (INSERT, UPDATE, or DELETE) occurs on a specified table or view.

When to use them:

- **Enforcing Complex Business Rules:** When standard constraints (Primary Key, Foreign Key, Check) are insufficient.
- **Auditing and Logging:** Recording changes to sensitive data (e.g., who changed what, when).
- **Maintaining Data Consistency:** Automatically updating related tables when changes occur in a primary table (e.g., updating stock quantity when an order is placed).
- **Replicating Data:** Propagating changes to other tables or databases.
- **Preventing Invalid Operations:** Rolling back transactions if certain conditions are not met.

Types (SQL Server): `FOR`/`AFTER` triggers (most common, fire after the DML operation), `INSTEAD OF` triggers (fire instead of the DML operation).

- *Note: MySQL supports `BEFORE` and `AFTER` triggers for `INSERT`, `UPDATE`, `DELETE` operations.*

Caution: Overuse of triggers can make database logic difficult to debug and maintain, and can lead to performance issues.

---

5. Concurrency and Locking

17. Explain database concurrency and the problems it can cause.

Database concurrency refers to the ability of a database to allow multiple users or processes to access and modify the same data simultaneously. While essential for multi-user systems, it can lead to several problems if not managed correctly:

- **Lost Updates:** When two transactions read the same data, modify it, and then write it back, the update of the first transaction can be overwritten by the second, leading to lost data.
- **Dirty Reads (Uncommitted Dependency):** When a transaction reads data that has been modified by another transaction but not yet committed. If the modifying transaction then rolls back, the data read by the first transaction is invalid.

- **Non-Repeatable Reads:** When a transaction reads the same row multiple times, but between reads, another committed transaction modifies or deletes that row, causing the first transaction to see different values.
- **Phantom Reads:** When a transaction executes a query that returns a set of rows, and then a second committed transaction inserts new rows that satisfy the first query's `WHERE` clause. When the first transaction re-executes the query, it sees "phantom" new rows.

18. How do databases handle concurrency? Discuss locking and transaction isolation levels.

Databases use **locking** and **transaction isolation levels** to manage concurrency and prevent the problems mentioned above.

- **Locking:**
  - **Purpose:** Mechanisms to restrict access to a data resource (row, page, table) while a transaction is using it, preventing other transactions from interfering.
  - **Types:**
    - **Shared Locks (Read Locks):** Allow multiple transactions to read the same resource concurrently. Prevent exclusive locks from being placed.
    - **Exclusive Locks (Write Locks):** Prevent any other transaction from reading or writing to the resource. Applied during update, insert, or delete operations.
- **Transaction Isolation Levels:** ANSI SQL defines four standard isolation levels, which control the degree to which one transaction's uncommitted changes are visible to other transactions. Higher isolation levels provide more data integrity but can reduce concurrency and increase locking overhead.
  - **`READ UNCOMMITTED` (Least Strict):** Allows dirty reads, non-repeatable reads, and phantom reads. Highest concurrency, lowest data integrity.
  - **`READ COMMITTED` (Default for SQL Server, Oracle):** Prevents dirty reads. Allows non-repeatable reads and phantom reads. Reads only committed data.
  - **`REPEATABLE READ` (Default for MySQL InnoDB):** Prevents dirty reads and non-repeatable reads. Allows phantom reads. Ensures that if you read a row multiple times within a transaction, you get the same values.
  - **`SERIALIZABLE` (Most Strict):** Prevents dirty reads, non-repeatable reads, and phantom reads. Simulates serial execution of transactions, meaning transactions execute as if there were no concurrency. Lowest concurrency, highest data integrity.
  - **Snapshot Isolation (SQL Server):** An alternative, row-versioning based isolation level that largely avoids locking for read operations, using row versions instead. Reduces blocking.

---

6. SQL Server / MySQL Specifics

19. What are the main differences between SQL Server and MySQL?

Feature	SQL Server (Microsoft)	MySQL (Oracle)
Licensing	Commercial (various editions), Free (Express)	Open Source (Community), Commercial (Enterprise)
Platform	Primarily Windows, now Linux, Docker	Windows, Linux, macOS, various Unix
Programming Language	T-SQL (Transact-SQL)	SQL, specific extensions
Default Port	1433	3306
Security	Integrated Windows Auth, SQL Auth, AD auth	MySQL user accounts, plugin-based auth
Management Tool	SQL Server Management Studio (SSMS)	MySQL Workbench, phpMyAdmin, command line
Indexing	Clustered & Non-Clustered	Clustered PKs (InnoDB), Non-Clustered
Storage Engines	Not applicable (single engine)	Pluggable (InnoDB, MyISAM, NDB, etc.)
Scalability	Good for large enterprises, data warehousing	Excellent for web-scale apps, high reads
Full-Text Search	Built-in	Built-in
High Availability	AlwaysOn Availability Groups, Failover Clusters	Replication (Master-Slave, Group Replication)

20. Explain the concept of Storage Engines in MySQL.

MySQL uses a **pluggable storage engine architecture**, which means the underlying data storage mechanism can be changed for different tables. This is a unique and powerful feature of MySQL, allowing you to choose an engine optimized for specific needs.

Key Storage Engines:

- `InnoDB` **(Default and Recommended):**
  - **Features:** Supports ACID transactions, row-level locking, foreign key constraints, crash recovery, and B-tree indexes.

  - **Use Cases:** General-purpose transactional applications requiring high reliability and concurrency.
- `MyISAM` **(Older, not transactional):**
  - **Features:** Faster for read-heavy operations, supports full-text indexing, table-level locking only, no transactions, no foreign keys, faster for `SELECT COUNT(*)` without a `WHERE` clause.
  - **Use Cases:** Web applications that are primarily read-only (e.g., simple blogs, static content sites). Not suitable for applications requiring data integrity or concurrency.
- **Other Engines:** `Memory` (for in-memory tables), `CSV` (for CSV files), `Archive` (for archiving data), `NDB` (for MySQL Cluster), etc.

You specify the storage engine when you create a table:

SQL
```
CREATE TABLE MyTransactionalTable (
    id INT PRIMARY KEY,
    name VARCHAR(100)
) ENGINE = InnoDB;

CREATE TABLE MyReadOnlyTable (
    id INT PRIMARY KEY,
    description TEXT
) ENGINE = MyISAM;
```

21. What is a CTE (Common Table Expression) in SQL Server?

A **Common Table Expression (CTE)** is a named, temporary result set that you can reference within a single SQL statement (SELECT, INSERT, UPDATE, DELETE, or CREATE VIEW). It's essentially a temporary view that lives only for the duration of the query.

Advantages:

- **Readability:** Breaks down complex queries into logical, readable steps.
- **Recursion:** Enables recursive queries (e.g., for hierarchical data like organizational charts).
- **Modularity:** A CTE can be referenced multiple times within the same query, avoiding redundant code.
- **Simplicity:** Can simplify complex subqueries and derived tables.

Syntax:
SQL
```
WITH CTE_Name (Column1, Column2, ...) AS (
    -- Your SELECT statement that defines the CTE
    SELECT ...
    FROM ...
    WHERE ...
)
-- Your main query that uses the CTE
SELECT ...
FROM CTE_Name
WHERE ...;
```

Example (Find employees with highest salary in each department):

SQL
```
WITH DepartmentMaxSalary AS (
    SELECT
        DepartmentID,
        MAX(Salary) AS MaxSalary
    FROM
        Employees
    GROUP BY
        DepartmentID
)
SELECT
    e.EmployeeID,
    e.Name,
    e.Salary,
    e.DepartmentID
FROM
    Employees e
JOIN
    DepartmentMaxSalary dms ON e.DepartmentID = dms.DepartmentID AND
e.Salary = dms.MaxSalary;
```

22. How do you perform database backups and restores in SQL Server/MySQL?

SQL Server:

- **Backup:**
  - **Full Backup:** Backs up the entire database.
  - **Differential Backup:** Backs up only the data that has changed since the last full backup.
  - **Transaction Log Backup:** Backs up the transaction log, allowing point-in-time recovery.
  - **Commands:** `BACKUP DATABASE YourDatabase TO DISK = 'C:\path\YourDatabase.bak';`
  - **Tools:** SQL Server Management Studio (SSMS) GUI, `sqlcmd` utility, SQL Server Agent jobs.
- **Restore:**
  - **Commands:** `RESTORE DATABASE YourDatabase FROM DISK = 'C:\path\YourDatabase.bak';`
  - **Tools:** SSMS GUI.

MySQL:

- **Backup:**
  - **`mysqldump` (Logical Backup):** Creates SQL statements that can recreate the database. Excellent for portability.
  - `mysqldump -u username -p database_name > backup.sql`
  - **Physical Backup:** Copying data files directly (e.g., using `percona-xtrabackup` for InnoDB).
  - **MySQL Workbench:** Provides GUI tools for backup.
- **Restore:**
  - **Using `mysql` client:** `mysql -u username -p database_name < backup.sql`
  - **MySQL Workbench:** GUI tools for restore.

Okay, let's cover common Angular interview questions and answers, focusing on core concepts, components, services, routing, and more.

---

1. Core Concepts

1. What is Angular and what are its key features?

Angular is an open-source, TypeScript-based front-end web application framework led by the Angular Team at Google and a community of individuals and corporations. It's used for building dynamic, single-page applications (SPAs).

Key Features:

- **Components:** The fundamental building blocks of Angular applications, encapsulating UI and logic.
- **TypeScript:** A superset of JavaScript that provides static typing, improving code quality and maintainability.
- **Data Binding:** Synchronizes data between the component's logic and the view (HTML template).
- **Directives:** Extend HTML with new behaviors.
- **Services & Dependency Injection:** Promotes reusable code and loose coupling by providing dependencies.
- **Routing:** Enables navigation between different views within an SPA without full page reloads.
- **RxJS (Reactive Extensions for JavaScript):** Used for handling asynchronous data streams.
- **Modular Architecture:** Organizes code into distinct, reusable modules.
- **CLI (Command Line Interface):** A powerful tool for creating, developing, and deploying Angular applications.

2. Explain the architectural components of an Angular application.

An Angular application is structured around a modular architecture, primarily consisting of:

- **Modules (`NgModule`):** A fundamental building block that organizes an application into cohesive blocks of functionality. Every Angular app has at least one root module (`AppModule`). Modules declare components, services, pipes, and directives that belong to them and import other modules whose functionalities they need.
- **Components:** The most basic UI building block. A component is a TypeScript class with an associated HTML template and a CSS stylesheet. It controls a portion of the screen.
- **Templates:** HTML views that define the layout and appearance of components. They use Angular's template syntax for data binding and directives.
- **Metadata:** Decorators (`@Component`, `@NgModule`, `@Injectable`) used to configure classes and specify their purpose in the Angular ecosystem.
- **Data Binding:** The communication mechanism between the component and its template. (See question 3)

- **Directives:** Classes that add extra behavior to elements in Angular templates. (See question 5)
- **Services:** Classes that encapsulate reusable logic and data access. They are typically injected into components or other services using Dependency Injection.
- **Dependency Injection (DI):** A design pattern where a class receives its dependencies from external sources rather than creating them itself. Angular's DI system manages the creation and provision of services.

3. What are the different types of Data Binding in Angular?

Data Binding is the synchronization of data between the **Model (component's logic)** and the **View (HTML template)**. Angular supports four main types:

- **1. Interpolation (`{{ }}`) - One-Way (Component to View):**
  - Displays component property values in the template.
  - Example: `<h1>Welcome, {{ userName }}!</h1>`
- **2. Property Binding (`[property]`) - One-Way (Component to View):**
  - Binds a component property to an HTML element's property.
  - Example: `<img [src]="imageUrl">`, `<button [disabled]="isButtonDisabled">Click Me</button>`
- **3. Event Binding (`(event)`) - One-Way (View to Component):**
  - Responds to events raised by HTML elements (e.g., `click`, `keyup`) and executes a component method.
  - Example: `<button (click)="submitForm()">Submit</button>`
- **4. Two-Way Data Binding (`[(ngModel)]`) - Two-Way (Component to View & View to Component):**
  - Combines property binding and event binding to allow data flow in both directions. Changes in the input field update the component property, and changes in the component property update the input field.
  - Requires `FormsModule` to be imported in the module.
  - Example: `<input [(ngModel)]="searchQuery">`

4. What is TypeScript and why does Angular use it?

TypeScript is an open-source superset of JavaScript that compiles to plain JavaScript. It adds static typing and object-oriented features to JavaScript.

Why Angular uses it:

- **Static Typing:** Allows developers to define types for variables, function parameters, and return values. This catches common programming errors at **compile time** rather than runtime, leading to more robust and maintainable code.
- **Enhanced Tooling:** Provides better IDE support, including IntelliSense, autocompletion, and refactoring capabilities, improving developer productivity.
- **Object-Oriented Features:** Supports classes, interfaces, inheritance, and access modifiers, enabling cleaner and more organized code, especially for large applications.

- **Readability and Maintainability:** Strong typing makes code easier to understand and refactor for teams.
- **Future JavaScript Features:** TypeScript often includes features from future ECMAScript versions before they are widely supported in browsers, allowing developers to use them today.

5. Explain Directives. What are the three types of directives?

Directives are classes that add extra behavior to elements in Angular templates. They are essentially markers on a DOM element that tell Angular to do something to that DOM element or its children.

There are three types of directives:

- **1. Component Directives:** These are directives with a template. In fact, **components are directives**! They are the most common type and define the main building blocks of the UI.
  - Example: `@Component({ selector: 'app-my-component', template: '...', styles: '...' })`
- **2. Structural Directives:** These directives change the **structure of the DOM** by adding, removing, or manipulating elements and their children. They are prefixed with an asterisk (`*`).
  - Examples:
    - *ngIf: Conditionally adds or removes an element from the DOM.

      <div *ngIf="showContent">Content is visible</div>

    - *ngFor: Renders a list of items by iterating over a collection.

      <li *ngFor="let item of items">{{ item }}</li>

    - `*ngSwitchCase`: Renders content based on a match with a switch value.
- **3. Attribute Directives:** These directives change the **appearance or behavior** of an existing DOM element. They typically modify the style or attributes of elements.
  - Examples:
    - NgStyle: Dynamically sets CSS styles.

      <div [ngStyle]="{'background-color': 'blue'}"></div>

    - NgClass: Dynamically adds or removes CSS classes.

      <div [ngClass]="{'active': isActive, 'highlight': isHighlight}"></div>

    - A custom attribute directive to highlight an element on hover.

---

2. Components and Templates

6. What is the difference between `ng-content`, `ng-template`, and `ng-container`?

These are special Angular elements/attributes used for content projection and template manipulation:

- `ng-content` **(Content Projection/Transclusion):**
  - Used in a **component's template** to define a placeholder where content from the **parent component** will be projected.
  - Allows you to create reusable components that can accept and display arbitrary content from their consumers.
  - Can use `select` attribute (e.g., `select="header"`, `select=".footer"`) to project specific content based on CSS selectors.
  - Example:

```html
HTML
<div class="card">
    <header><ng-content select="header"></ng-content></header>
    <div class="card-body"><ng-content></ng-content></div>
    <footer><ng-content select="footer"></ng-content></footer>
</div>
<app-reusable-card>
    <h2 header>Card Title</h2>
    <p>This is the main content of the card.</p>
    <button footer>Action</button>
</app-reusable-card>
```

- `ng-template`**:**
  - A template element that **Angular renders without attaching it to the DOM initially**. It's a way to define a block of HTML that can be rendered conditionally or multiple times.
  - Often used with structural directives like `*ngIf` with `else` block, `*ngFor` with `template`, or when creating custom structural directives.
  - Example with `*ngIf`:

```html
HTML
<div *ngIf="isLoggedIn; else loginBlock">
    Welcome, User!
</div>
<ng-template #loginBlock>
    Please log in.
</ng-template>
```

- `ng-container`**:**
  - A grouping element that **Angular does not add to the DOM**. It's rendered as a comment node in the DOM.
  - Useful when you need to apply a structural directive (e.g., `*ngIf`, `*ngFor`) to multiple elements without introducing an extra DOM element (like a `<div>` or `<span>`).
  - Example:

HTML
```
<table>
    <ng-container *ngFor="let item of items">
        <tr><td>{{ item.id }}</td></tr>
        <tr><td>{{ item.name }}</td></tr>
    </ng-container>
</table>
```

Without `ng-container`, you'd have to wrap `<tr>`s in an extra `div`, which is invalid HTML inside a `<table>`.

7. Explain `@Input()` and `@Output()` decorators.

These decorators are crucial for **component communication** in Angular:

- **`@Input()`:**
  - Used to define a **property that can receive data from a parent component**.
  - It makes a component property **bindable** from the outside.
  - Data flows **down** from parent to child.
  - Example:

TypeScript
```
// child.component.ts
import { Component, Input } from '@angular/core';

@Component({...})
export class ChildComponent {
    @Input() message: string; // Declares 'message' as an input property
}
```

HTML
```
<app-child [message]="'Hello from Parent!'"></app-child>
```

- **`@Output()`:**
  - Used to define an **event emitter** that allows a **child component to send data or notify the parent component** about an event.
  - Typically used with `EventEmitter` to emit custom events.
  - Data flows **up** from child to parent.
  - Example:

TypeScript
```
// child.component.ts
import { Component, Output, EventEmitter } from '@angular/core';

@Component({...})
export class ChildComponent {
    @Output() itemAdded = new EventEmitter<string>(); // Declares an
output event
```

```
    addItem() {
        this.itemAdded.emit('New Item Added'); // Emits the event
    }
}
```

HTML
```
<app-child (itemAdded)="onItemAddedHandler($event)"></app-child>
```

TypeScript
```
// parent.component.ts
onItemAddedHandler(data: string) {
    console.log('Received from child:', data);
}
```

8. What are Lifecycle Hooks in Angular? Name some common ones.

Lifecycle hooks are special methods that Angular calls on components and directives at specific points in their lifecycle. They allow you to tap into these events to perform initialization, update, and cleanup logic.

Common Lifecycle Hooks (in typical order of execution for a component):

- **`ngOnChanges()`:**
  - Called when Angular sets or resets data-bound input properties.
  - Receives a `SimpleChanges` object containing current and previous property values.
  - Useful for reacting to changes in `@Input()` properties.
- **`ngOnInit()`:**
  - Called once, after the component's input properties have been initialized for the first time.
  - **Best place for initial data retrieval (e.g., fetching data from a service), and complex initialization logic.**
- **`ngDoCheck()`:**
  - Called during every change detection cycle, immediately after `ngOnChanges` and `ngOnInit`.
  - Used for implementing custom change detection logic, especially when Angular's default change detection isn't sufficient (e.g., for objects not directly referenced by inputs).
- **`ngAfterContentInit()`:**
  - Called once after Angular has projected any content into the component's view.
  - Used to access elements projected via `ng-content` using `@ContentChild` or `@ContentChildren`.
- **`ngAfterContentChecked()`:**
  - Called after the content (projected via `ng-content`) has been checked during every change detection cycle.
- **`ngAfterViewInit()`:**
  - Called once after Angular initializes the component's views and child views.
  - **Best place to access child components/elements (from the component's own template) via `@ViewChild` or `@ViewChildren`.**

- `ngAfterViewChecked()`:
  - Called after the component's view and child views have been checked during every change detection cycle.
- `ngOnDestroy()`:
  - Called just before Angular destroys the component or directive.
  - **Essential for cleanup activities** to prevent memory leaks (e.g., unsubscribing from observables, detaching event handlers, clearing timers).

---

3. Services and Dependency Injection

9. What is a Service in Angular and why are they used?

A **Service** in Angular is a plain TypeScript class that encapsulates reusable business logic, data access, or utility functions that are not specific to a particular UI component.

Why they are used:

- **Separation of Concerns:** Helps keep components lean and focused on UI presentation. Logic is moved to services.
- **Reusability:** Services can be injected into multiple components or other services, avoiding code duplication.
- **Maintainability:** Easier to maintain and test code when logic is separated.
- **Data Sharing:** Provides a centralized way to share data across different parts of the application.
- **Testability:** Services can be easily mocked or stubbed for unit testing.

Services are typically decorated with `@Injectable()`.

10. Explain Dependency Injection (DI) in Angular.

Dependency Injection (DI) is a core design pattern in Angular. It's a way of providing required dependencies (like services) to a class (like a component or another service) rather than having the class create them itself.

How it works in Angular:

1. **Providers:** You register services (or other dependencies) with an **injector**. This is typically done in an `@NgModule` (`providers` array) or directly on the service using `providedIn: 'root'`.
   - `@Injectable({ providedIn: 'root' })` makes the service a singleton available throughout the app.
   - `providers: [MyService]` in a module makes it available to components/services declared within that module.
2. **Injectors:** Angular creates a hierarchical tree of injectors. The root injector provides application-wide singletons. Child modules/components can have their own injectors.

3. **Consumption:** When a component or service needs a dependency, it declares it in its **constructor parameters** with type hinting.
4. **Injection:** Angular's DI system inspects the constructor, finds the requested dependency in the nearest injector, creates an instance if one doesn't exist (based on its provider configuration), and provides it to the class.

Benefits of DI:

- **Loose Coupling:** Components don't know how their dependencies are created, making them independent.
- **Testability:** Easier to unit test components by injecting mock versions of services.
- **Reusability:** Services can be shared across the application.
- **Scalability:** Facilitates building larger, more complex applications.

Example:
TypeScript
```
// data.service.ts
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root' // Makes DataService a singleton available globally
})
export class DataService {
  constructor(private http: HttpClient) { } // HttpClient is injected

  getData(): Observable<any[]> {
    return this.http.get<any[]>('/api/data');
  }
}

// my.component.ts
import { Component, OnInit } from '@angular/core';
import { DataService } from './data.service';

@Component({ ... })
export class MyComponent implements OnInit {
  items: any[];

  constructor(private dataService: DataService) { } // DataService is
injected

  ngOnInit() {
    this.dataService.getData().subscribe(data => {
      this.items = data;
    });
  }
}
```

4. Routing and Navigation

11. What is Angular Routing and how do you configure it?

Angular Routing allows you to navigate between different views (components) within a single-page application without requiring full page reloads. It maps URLs to specific components, providing a seamless user experience.

Configuration Steps:

1. **Import `RouterModule` and `Routes`:** In your main routing module (e.g., `app-routing.module.ts`), import these.
2. **Define Routes Array:** Create an array of `Routes` objects, where each object defines a path and its corresponding component.
3. **Import/Export `RouterModule.forRoot()`/`forChild()`:**
   - Use `RouterModule.forRoot(routes)` in your **root routing module** (`AppRoutingModule`) to register top-level routes and configure the router.
   - Use `RouterModule.forChild(routes)` in **feature modules** (lazy-loaded modules) to register their specific routes.
4. **Add `router-outlet`:** Place `<router-outlet></router-outlet>` in your `AppComponent`'s template (or the template of any component where you want the routed components to be displayed). This is where Angular renders the activated component.
5. **Use `routerLink`:** Use the `routerLink` directive in your templates to create navigation links.

Example (`app-routing.module.ts`):
TypeScript
```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';
import { ContactComponent } from './contact/contact.component';
import { ProductListComponent } from './products/product-list/product-
list.component';
import { ProductDetailComponent } from './products/product-detail/product-
detail.component';
import { NotFoundComponent } from './not-found/not-found.component';

const routes: Routes = [
  { path: '', component: HomeComponent }, // Default route
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
  { path: 'products', component: ProductListComponent },
  { path: 'products/:id', component: ProductDetailComponent }, // Route
with parameter
  { path: '**', component: NotFoundComponent } // Wildcard route for 404
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Example (`app.component.html`):
HTML

````html
<nav>
  <a routerLink="/">Home</a> |
  <a routerLink="/about">About</a> |
  <a routerLink="/products">Products</a>
</nav>
<router-outlet></router-outlet> ```
````

~~~
### 12. How do you pass data between routes in Angular?
You can pass data between routes using several methods:

* **1. Route Parameters (URL segments):**
    * **When to use:** When data is essential for identifying the specific
resource on the target page (e.g., product ID, user ID).
    * **How:** Define a parameter in the route config (`path:
'products/:id'`). Access it in the target component using
`ActivatedRoute`.
        ```typescript
        // product-detail.component.ts
        import { ActivatedRoute } from '@angular/router';
        // ...
        constructor(private route: ActivatedRoute) {}

        ngOnInit() {
            this.route.paramMap.subscribe(params => {
                const productId = +params.get('id'); // '+' converts
string to number
                // Fetch product data using productId
            });
        }
        ```

* **2. Query Parameters:**
    * **When to use:** For optional parameters, filtering, or sorting
(e.g., `products?category=electronics&sort=price`).
    * **How:** Append `queryParams` to `routerLink` or use
`Maps()`/`MapsByUrl()`. Access in target component using `ActivatedRoute`.
        ```html
        <a [routerLink]="['/products']" [queryParams]="{ category:
'electronics', sort: 'price' }">Electronics</a>
        ```
        ```typescript
        // product-list.component.ts
        ngOnInit() {
            this.route.queryParamMap.subscribe(params => {
                const category = params.get('category');
                const sort = params.get('sort');
                // Filter products based on category and sort
~~~

            });
        }

* **3. State Object (Router NavigationExtras):**
    * **When to use:** For passing complex or sensitive data that shouldn't appear in the URL, especially during programmatic navigation. The data is available only for the *initial navigation* and not after a page refresh.
    * **How:** Pass a `state` object in the `Router.navigate()` method's `NavigationExtras`. Access in target component via `Router.getCurrentNavigation()`.
        ```typescript
        // Source Component:
        constructor(private router: Router) {}
        goToDetail() {
            this.router.navigate(['/product-detail'], {
                state: { productData: { id: 1, name: 'Laptop' } }
            });
        }
        ```
        ```typescript
        // Target Component:
        constructor(private router: Router) {}
        ngOnInit() {
            if (this.router.getCurrentNavigation()?.extras.state) {
                const product =
this.router.getCurrentNavigation().extras.state['productData'];
                console.log(product);
            }
        }
        ```

* **4. Services (Preferred for persistent or shared data):**
    * **When to use:** For data that needs to persist across multiple routes, or data that is shared between unrelated components.
    * **How:** Create an injectable service, store the data in it (e.g., using an RxJS `BehaviorSubject`), and inject the service into both the source and target components to share the data.

---

5. Forms

13. What is the difference between Template-Driven Forms and Reactive Forms?
Angular offers two main approaches for building forms:

Feature	Template-Driven Forms	Reactive Forms (Model-Driven)
Setup	Minimal code in component, logic in template	Logic primarily in component class

Form Model	Created implicitly by directives (`ngModel`)	Explicitly created in component (`FormControl`, `FormGroup`, `FormArray`)
Control Creation	Automatically created by Angular directives	Programmatic, imperative
Validation	Primarily uses `data-` attributes/HTML5 validators or custom validators with directives (`ngModelGroup`)	Validator functions passed explicitly to `FormControl`
Scalability	Good for simple forms	Ideal for complex, dynamic forms
Testability	Harder to unit test	Easier to unit test
Immutability	Mutable data model	Immutable data model (more reactive)
Module	`FormsModule`	`ReactiveFormsModule`

When to use which:

* **Template-Driven Forms:**
    * Simple forms with basic validation.
    * When you prefer to build most of your form logic directly in the template using directives.
    * Quick prototyping.

* **Reactive Forms:**
    * Complex forms with dynamic fields, conditional validation, or custom validation logic.
    * When you need more control over form validation and submission.
    * When you need to unit test your form logic rigorously.
    * Preferred for larger, enterprise-level applications due to their explicit, testable, and scalable nature.

14. How do you implement custom validation in Angular forms?
You can implement custom validation for both Template-Driven and Reactive Forms.

1. For Reactive Forms (Recommended):
* Create a function that returns a `ValidationErrors` object (if invalid) or `null` (if valid).
* The function takes an `AbstractControl` (or `FormControl`, `FormGroup`, `FormArray`) as an argument.
* Assign the validator function directly to your `FormControl` or `FormGroup`.

    ```typescript
    import { AbstractControl, ValidatorFn, ValidationErrors } from '@angular/forms';

    // Custom validator for reactive forms
    export function forbiddenNameValidator(nameRe: RegExp): ValidatorFn {
      return (control: AbstractControl): ValidationErrors | null => {
        const forbidden = nameRe.test(control.value);
    ```

```typescript
        return forbidden ? { forbiddenName: { value: control.value } } : null;
      };
    }

    // In your component:
    this.heroForm = new FormGroup({
      'name': new FormControl(this.hero.name, [
        Validators.required,
        Validators.minLength(4),
        forbiddenNameValidator(/bob/i) // Using custom validator
      ]),
      // ...
    });
```

2. For Template-Driven Forms:
* Create a **custom directive** that implements the `Validator` interface.
* Register the directive with Angular's `NG_VALIDATORS` token.

```typescript
    import { Directive, Input } from '@angular/core';
    import { AbstractControl, NG_VALIDATORS, Validator, ValidationErrors } from '@angular/forms';

    @Directive({
      selector: '[appForbiddenName]',
      providers: [{provide: NG_VALIDATORS, useExisting: ForbiddenValidatorDirective, multi: true}]
    })
    export class ForbiddenValidatorDirective implements Validator {
      @Input('appForbiddenName') forbiddenName: string; // Input to pass the forbidden name

      validate(control: AbstractControl): ValidationErrors | null {
        const forbidden = new RegExp(this.forbiddenName, 'i').test(control.value);
        return forbidden ? { 'forbiddenName': { value: control.value } } : null;
      }
    }
```
```html
    <input type="text" id="heroName" name="heroName"
           [(ngModel)]="hero.name"
           appForbiddenName="bob" required>
    <div *ngIf="name.errors?.forbiddenName">Name cannot be Bob.</div>
```

---

6. RxJS and State Management

15. What is RxJS and Observables in Angular? Why are they used?

* **RxJS (Reactive Extensions for JavaScript):** A library for **reactive programming** using **Observables**, to make it easier to compose asynchronous or callback-based code sequences.
* **Observable:** Represents a stream of data that can be emitted over time. It's a "push system" where the Observable pushes data to its subscribers. Observables are lazy, meaning they don't start emitting values until someone subscribes to them.

Why they are used in Angular:
* **Asynchronous Operations:** Angular uses Observables extensively for handling asynchronous events like:
    * HTTP requests (`HttpClient` returns Observables).
    * Router events.
    * Form control value changes.
    * Event handling (e.g., from `RxJs.fromEvent`).
* **Reactive Programming:** Enables a more declarative and functional approach to handling complex asynchronous data flows, making code cleaner and easier to reason about.
* **Powerful Operators:** RxJS provides a rich set of operators (e.g., `map`, `filter`, `debounceTime`, `switchMap`, `mergeMap`, `takeUntil`) for transforming, combining, and manipulating data streams.
* **Error Handling:** Provides robust mechanisms for handling errors in asynchronous operations.
* **Resource Management:** Observables can be easily "unsubscribed" from to prevent memory leaks.

Example:
```typescript
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { map, catchError } from 'rxjs/operators'; // Operators

@Component({...})
export class DataDisplayComponent implements OnInit {
  data$: Observable<any[]>; // Convention: suffix with '$' for Observables

  constructor(private http: HttpClient) { }

  ngOnInit() {
    this.data$ = this.http.get<any[]>('/api/data').pipe(
      map(response => response.filter(item => item.isActive)), //
Transform data
      catchError(error => {
        console.error('Error fetching data:', error);
        return []; // Return empty array or throw an error
      })
    );
  }

  // In template, using async pipe:
  // <div *ngIf="data$ | async as data">
  //   <li *ngFor="let item of data">{{ item.name }}</li>
  // </div>
}
```

16. What is the `async` pipe in Angular?

The `async` pipe (`| async`) is a powerful built-in Angular pipe that simplifies working with `Observables` (and `Promises`) in templates.

How it works:

- It **subscribes** to an `Observable` (or `Promise`) and unwraps the emitted value.
- When a new value is emitted, it automatically triggers **change detection** to update the view.
- Crucially, when the component is destroyed, the `async` pipe **automatically unsubscribes** from the Observable, preventing memory leaks.

Benefits:

- **Simplified Template Code:** No need to manually subscribe/unsubscribe in the component class.
- **Memory Leak Prevention:** Automatically handles unsubscribing.
- **Automatic Change Detection:** Updates the view whenever a new value arrives.

Example:
TypeScript
```
// data-component.ts
import { Component } from '@angular/core';
import { Observable, of } from 'rxjs';

@Component({
  selector: 'app-data',
  template: `
    <div *ngIf="data$ | async as data">
      <p>Data loaded: {{ data }}</p>
    </div>
    <p *ngIf="time$ | async as currentTime">Current Time: {{ currentTime |
date:'mediumTime' }}</p>
  `
})
export class DataComponent {
  data$: Observable<string> = of('Hello Async Pipe!');
  time$: Observable<Date> = new Observable(observer => {
    setInterval(() => observer.next(new Date()), 1000);
  });
}
```

---

7. Change Detection

17. Explain Angular's Change Detection mechanism.

Change Detection is the process by which Angular detects changes in component data and re-renders the DOM to reflect those changes. It's how the view stays in sync with the model.

How it works (Simplified):

1. Angular builds a **change detector tree** for all components in the application.
2. When an asynchronous event occurs (e.g., user interaction, HTTP response, timer, `setTimeout`), Angular's **Zone.js** intercepts it.
3. Zone.js notifies Angular that something might have changed.
4. Angular then traverses the change detector tree, typically from the root down.
5. For each component, it compares the current value of its bound properties (inputs, template variables) with their previous values.
6. If a change is detected, Angular updates the corresponding part of the DOM.

Default Strategy (`ChangeDetectionStrategy.Default`):

- Angular checks every component in the component tree during each change detection cycle, even if its inputs haven't changed. This is safe but can be inefficient for very large applications.

18. What is `OnPush` Change Detection Strategy and when would you use it?

`OnPush` Change Detection Strategy is an optimization technique that tells Angular to run change detection for a component (and its subtree) **only when its inputs change** or when an observable it is subscribed to emits a new value (when using the `async` pipe).

When Angular checks an `OnPush` component:

1. **Input property changes (by reference):** An `@Input()` property's **reference** changes (e.g., a new object is assigned, not just a mutation within the existing object).
2. **An event originates from the component or its children:** (e.g., a button click within the component).
3. **An `Observable` bound to the template via `async` pipe emits a new value.**
4. **Manually triggered:** When `ChangeDetectorRef.detectChanges()` or `ChangeDetectorRef.markForCheck()` is explicitly called.

When to use it:

- **Performance Optimization:** For components that are "pure" (meaning their output depends only on their inputs) or whose inputs rarely change. It significantly reduces the number of change detection checks, improving application performance, especially in large applications with many components.
- **Immutable Data:** Works best with immutable data structures, as changes are only detected when references change.
- **Components with `async` pipe:** The `async` pipe naturally handles `OnPush` components efficiently.

How to use:
TypeScript
```
import { Component, ChangeDetectionStrategy } from '@angular/core';

@Component({
  selector: 'app-my-optimized-component',
  templateUrl: './my-optimized-component.html',
  styleUrls: ['./my-optimized-component.css'],
  changeDetection: ChangeDetectionStrategy.OnPush // Apply OnPush strategy
})
export class MyOptimizedComponent {
  // ...
}
```

---

8. Modules

19. Explain Lazy Loading in Angular modules. Why is it beneficial?

Lazy Loading is an Angular feature that allows you to load parts of your application's code (specifically, feature modules) **on demand** (e.g., when a user navigates to a particular route) rather than loading everything upfront when the application starts.

How it's configured:

You define routes for lazy-loaded modules using loadChildren property in your routing configuration.

Example:
TypeScript
```
const routes: Routes = [
  // ... other routes
  {
    path: 'admin',
    loadChildren: () => import('./admin/admin.module').then(m =>
m.AdminModule)
  }
];
```

Benefits:

- **Faster Initial Load Times:** The main application bundle is smaller, so the application loads and becomes interactive much quicker. This is a significant performance improvement, especially for large applications.
- **Reduced Memory Consumption:** Only the code that is currently needed is loaded into memory.
- **Improved User Experience:** Users don't have to wait for the entire application to download before they can start interacting with it.

- **Better Organization:** Encourages modular development, leading to a more organized and maintainable codebase.

---

9. Pipes

20. What are Pipes in Angular? Give some examples.

Pipes are a way to transform data in Angular templates before it's displayed to the user. They take data as input and transform it into a desired output format without changing the original data. They are similar to filters in other frameworks.

Syntax: `{{ value | pipeName[:param1][:param2] }}`

Types:

- **Built-in Pipes:** Angular provides a rich set of built-in pipes.
- **Custom Pipes:** You can create your own custom pipes.

Examples of Built-in Pipes:

- **`DatePipe`:** Formats dates.
  - `{{ today | date }}` (e.g., "Jul 21, 2025")
  - `{{ today | date:'short' }}` (e.g., "7/21/25, 10:30 AM")
  - `{{ today | date:'fullDate' }}` (e.g., "Monday, July 21, 2025")
- **`CurrencyPipe`:** Formats numbers as currency.
  - `{{ price | currency }}` (e.g., "$25.99")
  - `{{ price | currency:'EUR':'symbol':'1.2-2' }}` (e.g., "€25.99")
- **`DecimalPipe`:** Formats numbers with a specific number of decimal places.
  - `{{ myNumber | number:'1.2-2' }}` (minIntDigits.minFractionDigits-maxFractionDigits)
- **`LowerCasePipe`** / **`UpperCasePipe`:** Converts text to lower or upper case.
  - `{{ 'Hello World' | lowercase }}` (e.g., "hello world")
- **`PercentPipe`:** Formats numbers as percentages.
  - `{{ 0.75 | percent }}` (e.g., "75%")
- **`JsonPipe`:** Converts a JavaScript object to a JSON string. Useful for debugging.
  - `{{ myObject | json }}`
- **`AsyncPipe`:** (Already discussed) Subscribes to an Observable or Promise and returns its latest value.

Creating a Custom Pipe:
TypeScript
```typescript
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'shorten'
})
```

```
export class ShortenPipe implements PipeTransform {
  transform(value: string, limit: number): string {
    if (value.length > limit) {
      return value.substring(0, limit) + '...';
    }
    return value;
  }
}
```

Usage: `{{ 'This is a long text' | shorten:10 }}` (Output: "This is a...")

---

10. Advanced/Miscellaneous

21. What are Interceptors in Angular? When would you use them?

HTTP Interceptors are a powerful feature in Angular that allows you to intercept and modify HTTP requests and responses globally, before they are sent to the server or before they reach your component. They implement the `HttpInterceptor` interface.

When to use them:

- **Authentication/Authorization:** Adding authentication tokens (e.g., JWT) to outgoing request headers.
- **Error Handling:** Catching global HTTP errors (e.g., 401 Unauthorized, 500 Internal Server Error) and displaying notifications or redirecting.
- **Logging:** Logging all HTTP requests and responses.
- **Caching:** Implementing client-side caching strategies.
- **Transforming Data:** Modifying request or response data (e.g., adding common headers, transforming API responses).
- **Loading Indicators:** Showing a global spinner or progress bar for all HTTP requests.

How to create and register an Interceptor:

1. **Create a Service:** Implement the `HttpInterceptor` interface, which requires the `intercept(req: HttpRequest<any>, next: HttpHandler)` method.

   TypeScript
   ```
   // auth.interceptor.ts
   import { Injectable } from '@angular/core';
   import { HttpInterceptor, HttpRequest, HttpHandler, HttpEvent } from
   '@angular/common/http';
   import { Observable } from 'rxjs';

   @Injectable()
   export class AuthInterceptor implements HttpInterceptor {
     intercept(req: HttpRequest<any>, next: HttpHandler):
   Observable<HttpEvent<any>> {
   ```

```
    const authToken = 'your-jwt-token'; // Get from local storage/service
    const authReq = req.clone({
      headers: req.headers.set('Authorization', `Bearer ${authToken}`)
    });
    return next.handle(authReq);
  }
}
```

2. **Register in `AppModule`:** Add it to the `providers` array using `HTTP_INTERCEPTORS` token and `multi: true`.

TypeScript
```
import { HTTP_INTERCEPTORS } from '@angular/common/http';
import { AuthInterceptor } from './auth.interceptor';

@NgModule({
  declarations: [...],
  imports: [...],
  providers: [
    { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true }
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

22. What is Ahead-of-Time (AOT) Compilation in Angular? Why is it important?

Ahead-of-Time (AOT) Compilation is a compilation strategy where Angular templates and components are compiled into JavaScript **during the build process** (before the browser downloads and runs the application). This is the default compilation mode for production builds in Angular CLI.

Difference from Just-in-Time (JIT) Compilation:

- **JIT:** Compilation happens in the browser **at runtime**. (Default for `ng serve` / development builds).

Why AOT is important and beneficial (especially for production):

- **Faster Rendering:** The browser downloads a pre-compiled version of the application, so it can render the application immediately without an extra compilation step at runtime.
- **Smaller Application Size:** The Angular compiler (`ngc`) is not shipped to the browser, significantly reducing the size of the downloaded application bundle. Unused code (dead code elimination) is also removed.
- **Fewer Asynchronous Requests:** HTML and CSS are inlined into the JavaScript, reducing the number of requests the browser needs to make.
- **Early Error Detection:** Template syntax errors are caught during the build process rather than at runtime in the user's browser, leading to more robust applications.

- **Enhanced Security:** No dynamic evaluation of templates, reducing potential for injection attacks.

23. Explain `ngZone` and its role in Change Detection.

`ngZone` (powered by Zone.js) is a crucial part of Angular's change detection mechanism. It creates a **"zone"** (an execution context) that wraps all asynchronous operations originating from the browser (e.g., `setTimeout`, `setInterval`, `XMLHttpRequest` (HTTP requests), DOM events like `click`, `keyup`).

Role in Change Detection:

1. When an asynchronous task starts, `ngZone` records it.
2. When an asynchronous task **completes** (e.g., an HTTP request returns, a click event finishes), `ngZone` is notified.
3. `ngZone` then triggers Angular's change detection cycle, prompting Angular to check for changes and update the UI.

This mechanism allows Angular to **automatically detect changes** without developers having to manually call change detection methods after every asynchronous operation.

When to interact with `ngZone` directly:

- **Running outside Angular's zone (`ngZone.runOutsideAngular(() => { ... })`):** For performance-critical, frequently occurring operations (e.g., intense calculations, third-party library animations) that do not directly affect Angular's data. This prevents unnecessary change detection cycles.
- **Running inside Angular's zone (`ngZone.run(() => { ... })`):** If you perform an operation outside Angular's zone but then need to update Angular-bound data and trigger change detection.

24. What are Web Workers and how can they improve Angular app performance?

Web Workers are a browser feature that allows you to run scripts in a **background thread**, separate from the main UI thread. This means computationally intensive tasks can be performed without blocking the main thread, keeping the user interface responsive.

How they improve Angular app performance:

- **UI Responsiveness:** Prevents UI freezes or stuttering by offloading heavy calculations, data processing, or large computations from the main thread.
- **Parallelism:** Allows for true parallel execution of JavaScript code.
- **Improved User Experience:** Ensures a smooth and fluid user experience, even during complex operations.

Use Cases in Angular:

- Heavy data manipulation or calculations (e.g., complex image processing, video rendering, encryption).
- Parsing large JSON files.
- Running machine learning algorithms in the browser.
- Any task that takes more than a few milliseconds and would otherwise block the UI.

Angular CLI Support: Angular CLI has built-in support for generating Web Workers, making it easier to integrate them into your application.

Limitations:

- Web Workers cannot directly access the DOM. Communication with the main thread is done via `postMessage()` and `onmessage` events.
- Limited browser support for all features (though core Web Workers are widely supported).

Let's break down Angular's `ng-content` (content projection) and `async` pipe.

`ng-content` (Content Projection)

1. What is Content Projection (`ng-content`) in Angular and why is it used?

Content Projection, also known as **Transclusion** in other frameworks, is an Angular mechanism that allows you to "project" or insert content from a parent component directly into a specific placeholder within a child component's template. The `ng-content` element serves as this placeholder.

Why it's used:

- **Reusability and Flexibility:** It enables you to create highly reusable and configurable components. Instead of hardcoding content into a child component, you design it to accept content dynamically from its parent. This is particularly useful for layout components (like cards, modals, panels) that provide a consistent structure but need to display varied inner content.
- **Separation of Concerns:** The child component focuses on its structural role (e.g., how a card is laid out), while the parent component is responsible for providing the actual content that fills that structure.
- **Composition:** It allows for building complex UIs by composing smaller, well-defined components.

Analogy: Think of it like a picture frame. The frame (child component) provides the structure and styling, but the picture (content projected from the parent) is what makes each frame unique. The `ng-content` is where you place the picture.

2. How do you implement content projection? Explain single-slot and multi-slot projection.

Content projection is implemented using the `<ng-content>` element in the child component's template.

Single-Slot Content Projection

This is the simplest form, where all projected content goes into a single `ng-content` placeholder.

Example:

TypeScript

```
// card.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-card',
  template: `
    <div class="card-container">
      <div class="card-header">
        <ng-content select=".card-title"></ng-content>
      </div>
      <div class="card-body">
        <ng-content></ng-content> </div>
    </div>
  `,
  styles: [`
    .card-container { border: 1px solid #ccc; padding: 10px; margin: 10px;
}
    .card-header { font-weight: bold; margin-bottom: 5px; }
  `]
})
export class CardComponent { }
```

HTML

```
<app-card>
  <h2 class="card-title">My Awesome Card</h2>
  <p>This is the main content for the card body.</p>
  <button>Click Me</button>
</app-card>
```

In this example, the `<p>` and `<button>` elements will be projected into the `ng-content` without a `select` attribute, while the `<h2>` will go into the one with `select=".card-title"`.

Multi-Slot Content Projection

This allows you to project different pieces of content into specific, named slots within the child component's template. You achieve this using the `select` attribute on `ng-content`. The `select` attribute accepts CSS selectors (tag names, class names, attribute names, IDs).

Example:

TypeScript

```
// layout-panel.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-layout-panel',
  template: `
    <div class="panel">
      <header>
        <ng-content select="[panel-header]"></ng-content>
      </header>
      <div class="content">
        <ng-content></ng-content> </div>
      <footer>
        <ng-content select="[panel-footer]"></ng-content>
      </footer>
    </div>
  `,
  styles: [`
    .panel { border: 2px solid #007bff; padding: 15px; margin: 15px; }
    header { background-color: #e9ecef; padding: 5px; margin-bottom: 10px;
}
    footer { border-top: 1px solid #dee2e6; padding-top: 10px; margin-top:
10px; }
  `]
})
export class LayoutPanelComponent { }
```

HTML

```
<app-layout-panel>
  <h3 panel-header>Welcome to the Dashboard</h3>
  <p>Here's some important information you should know.</p>
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
  </ul>
  <button panel-footer class="btn btn-primary">Go to Settings</button>
</app-layout-panel>
```

In this scenario:

- The `<h3>` with the `panel-header` attribute will go into the header `ng-content`.
- The `<p>` and `<ul>` elements will go into the default `ng-content` (without a `select` attribute).
- The `<button>` with the `panel-footer` attribute will go into the footer `ng-content`.

If a child component's template has multiple `ng-content` elements, the ones with `select` attributes will receive content matching their selectors. Any remaining content will be projected into the `ng-content` element *without* a `select` attribute (the default slot). If there's no default slot, the unselected content is discarded.

`async` Pipe

3. What is the `async` pipe in Angular and why is it beneficial?

The `async` pipe (`| async`) is a built-in Angular pipe that simplifies working with **Observables** (and Promises) directly within your templates. It's an indispensable tool for reactive programming in Angular.

How it works:

1. **Subscription:** When Angular evaluates a template expression containing the `async` pipe, it automatically **subscribes** to the Observable (or Promise) that you pipe into it.
2. **Unwrapping:** As the Observable emits new values, the `async` pipe unwraps them and makes them available for display in the template.
3. **Change Detection:** Whenever a new value is emitted, the `async` pipe automatically triggers **change detection** for the component, ensuring the UI updates to reflect the latest data.
4. **Automatic Unsubscription:** Crucially, when the component (or the view containing the `async` pipe) is destroyed, the `async` pipe **automatically unsubscribes** from the Observable. This prevents memory leaks that can occur if you manually subscribe in your component and forget to unsubscribe.

Why it's beneficial:

- **Simplified Template Code:** It eliminates the need to manually subscribe to Observables in your component's TypeScript code (`.subscribe()`) and store the data in a component property. This makes templates cleaner and more declarative.
- **Automatic Memory Management:** The automatic unsubscription feature is a significant benefit. Manually managing subscriptions can be error-prone and lead to memory leaks if `unsubscribe()` is not called in `ngOnDestroy()`. The `async` pipe handles this boilerplate for you.
- **Automatic Change Detection:** It seamlessly integrates with Angular's change detection mechanism, ensuring the view is always up-to-date with the latest data from the Observable.
- **Works well with `OnPush` Strategy:** When using the `OnPush` change detection strategy, the `async` pipe is a perfect fit. It ensures that the component only re-renders when the Observable it's bound to emits a new value, optimizing performance.
- **Error and Completion Handling:** While not explicitly shown in the pipe, the Observable itself can handle errors and completion, and the `async` pipe will reflect that by no longer rendering content if the Observable completes or errors out.

4. Provide examples of using the `async` pipe with `*ngIf` and `*ngFor`.

The `async` pipe is frequently used with structural directives like `*ngIf` and `*ngFor` to handle data that arrives asynchronously.

Example with `*ngIf`

This pattern is often used to display data only after it has been loaded, or to show a loading indicator while data is being fetched.

Component (`data-loader.component.ts`):
TypeScript

```
import { Component, OnInit } from '@angular/core';
import { Observable, of, delay } from 'rxjs';

@Component({
  selector: 'app-data-loader',
  template: `
    <h2>Async Pipe with *ngIf</h2>
    <p *ngIf="isLoading">Loading data...</p>
    <div *ngIf="data$ | async as loadedData">
      <p>Data loaded successfully:</p>
      <pre>{{ loadedData | json }}</pre>
    </div>
    <p *ngIf="!(data$ | async) && !isLoading">No data available.</p>
  `,
})
export class DataLoaderComponent implements OnInit {
  data$: Observable<any>;
  isLoading: boolean = true;

  ngOnInit() {
    // Simulate an HTTP request
    this.data$ = of({ id: 1, name: 'Sample Item', value: 123
}).pipe(delay(2000));

    // After a delay, set isLoading to false (can be combined with tap
operator)
    setTimeout(() => {
      this.isLoading = false;
    }, 2000);
  }
}
```

Explanation:

- `data$ | async as loadedData`: The `async` pipe subscribes to `data$`. `as loadedData` assigns the emitted value to a local template variable `loadedData`, which is then accessible within the `*ngIf` block.
- The content inside the `div` will only be rendered once `data$` emits a value.

Example with `*ngFor`

This is common for rendering lists of items fetched asynchronously.

Component (`product-list.component.ts`):

TypeScript

```
import { Component, OnInit } from '@angular/core';
import { Observable, of, delay } from 'rxjs';

interface Product {
  id: number;
  name: string;
  price: number;
}

@Component({
  selector: 'app-product-list',
  template: `
    <h2>Async Pipe with *ngFor</h2>
    <p *ngIf="!products$ | async">Loading products...</p>
    <ul *ngIf="products$ | async as products">
      <li *ngFor="let product of products">
        {{ product.name }} - {{ product.price | currency:'INR' }}
      </li>
    </ul>
    <p *ngIf="(products$ | async)?.length === 0">No products found.</p>
  `,
})
export class ProductListComponent implements OnInit {
  products$: Observable<Product[]>;

  ngOnInit() {
    // Simulate an HTTP request fetching a list of products
    const sampleProducts: Product[] = [
      { id: 1, name: 'Laptop', price: 75000 },
      { id: 2, name: 'Mouse', price: 1200 },
      { id: 3, name: 'Keyboard', price: 3500 },
    ];
    this.products$ = of(sampleProducts).pipe(delay(1500));
  }
}
```

Explanation:

- `products$ | async as products`: The `async` pipe subscribes to `products$`. Once it emits an array, that array is assigned to the local `products` variable.
- The `*ngFor` then iterates over the `products` array, rendering a `<li>` for each item.
- The initial `<p>` with "Loading products..." can be used to show a temporary message until data arrives.
- The last `<p>` checks if the array is empty after loading.

These examples highlight how the `async` pipe elegantly handles the lifecycle of Observables, making template code cleaner and safer by automatically managing subscriptions and change detection.