

.NET Unit - 3



Help me with questions of my subject named Web Technology with .NET Framework.



Sure, I can help you with questions related to Web Technology with the .NET Framework. Let me know the specific topics or questions you need assistance with.



Draw the diagram and explain cache read/write in computing. (Give me link to search for diagram do not generate one)



In computing, cache memory plays a critical role in improving processing efficiency by storing frequently accessed data close to the CPU. Here's how cache read and write operations work:

Cache Read

1. **Address Lookup:** The CPU sends a memory address to the cache.
2. **Tag Matching:** The cache checks the tag portion of the address to see if the requested data block is stored.
3. **Cache Hit:** If the data is in the cache, it is retrieved directly, saving time.
4. **Cache Miss:** If the data is not found, it is fetched from the main memory and stored in the cache for future use.

Cache Write

1. **Write-Through:** The data is written simultaneously to the cache and main memory, ensuring both are synchronized.
2. **Write-Back:** The data is written only to the cache. The main memory is updated later when the cache block is evicted, using a "dirty bit" to track modified blocks.

Diagram

You can find diagrams explaining this process in detail through the following resources:

- [Cache Read/Write Diagram on CS Swarthmore](#)[Swarthmore College Computer Science](#).
- [Baeldung's Introduction to Cache Memory](#)[Baeldung](#).

These diagrams visually illustrate cache hits, misses, and the write strategies used in modern computer architecture.



2. Discuss advantages and limitations of IMemoryCache vs IDistributedCache.



IMemoryCache vs. IDistributedCache

In .NET, **IMemoryCache** and **IDistributedCache** are caching mechanisms with distinct purposes. Here's a comparison of their **advantages** and **limitations**:

IMemoryCache

IMemoryCache stores data in memory on the same server where the application is running.

Advantages:

1. **High Performance:**

- Extremely fast because data is stored in memory (RAM) on the local machine.
- No network latency since it avoids external storage systems.

2. **Ease of Use:**

- Simple to implement and configure.
- Ideal for small-scale applications or short-lived data.

3. **Cost Efficiency:**

- No external storage cost since it's part of the application server.

Limitations:

1. **Non-Distributed:**

- Data is not shared between multiple servers in a web farm.
- Not suitable for multi-server environments or cloud applications where scaling is required.

2. **Data Loss:**

- Cached data is lost if the application restarts or the server crashes.

3. **Memory Usage:**

- Relies on server memory, which may impact application performance under high load.
-

IDistributedCache

IDistributedCache stores data in external systems like Redis, SQL Server, or Azure, making it accessible across servers.

Advantages:

1. **Scalability:**

- Works in distributed environments, ensuring cache consistency across multiple servers.
- Ideal for cloud-based or containerized applications.

2. **Persistence:**

- Data survives server restarts, depending on the external storage used.

3. Flexibility:

- Supports multiple back-end providers (e.g., Redis, SQL Server, NCache).

Limitations:

1. Higher Latency:

- Requires network access to retrieve data, which is slower than in-memory operations.

2. Complexity:

- Requires additional configuration and management of the external cache infrastructure.

3. Cost:

- Involves costs for external storage services like Redis or SQL Server.
-

When to Use:

• IMemoryCache:

- Best for single-server applications or scenarios where performance is critical, and data persistence is not required.

• IDistributedCache:

- Ideal for applications deployed in distributed or cloud environments where cache consistency and persistence matter.

For more information:

- Microsoft Documentation on [IMemoryCache](#)
- Microsoft Documentation on [IDistributedCache](#).



3. Develop ASP.NET Core MVC application and demonstrate use of data caching.



To demonstrate data caching in an **ASP.NET Core MVC application**, follow these steps:

Step 1: Create a New ASP.NET Core MVC Project

1. Open Visual Studio and create a new ASP.NET Core Web Application.
2. Select "Web Application (Model-View-Controller)" and configure the project.

Step 2: Add IMemoryCache to the Application

1. Open the ``Startup.cs`` (or ``Program.cs`` in .NET 6+) file and register **IMemoryCache** in the ``ConfigureServices`` method:

```
csharp

public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
    services.AddMemoryCache(); // Add IMemoryCache service
}
```

Step 3: Implement Caching in a Controller

Create a ``ProductController`` to demonstrate caching product data.

```
csharp

using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Caching.Memory;
using System;
using System.Collections.Generic;

public class ProductController : Controller
{
    private readonly IMemoryCache _cache;

    public ProductController(IMemoryCache memoryCache)
    {
        _cache = memoryCache;
    }

    public IActionResult Index()
    {
        string cacheKey = "productList";
        List<string> products;

        // Check if the cache contains the data
        if (!_cache.TryGetValue(cacheKey, out products))
        {
            // ... (Implementation for fetching products from database or API)
        }
    }
}
```

```
// Simulate fetching data from a database
products = new List<string> { "Laptop", "Smartphone", "Tablet", "Smartwatch" };

// Set cache options
var cacheEntryOptions = new MemoryCacheEntryOptions()
    .SetSlidingExpiration(TimeSpan.FromSeconds(30)) // Cache expires if not accessed for 30 seconds
    .SetAbsoluteExpiration(TimeSpan.FromMinutes(2)); // Cache expires after 2 minutes

// Save data in cache
_cache.Set(cacheKey, products, cacheEntryOptions);
}

return View(products);
}
```

Step 4: Create the View

Create a `Views/Product/Index.cshtml` file to display the cached product list.

```
html

@model List<string>

<h2>Product List</h2>
<ul>
    @foreach (var product in Model)
    {
        <li>@product</li>
    }
</ul>
```

Step 5: Run the Application

1. Build and run the application.
2. Navigate to `/Product/Index`.
 - o On the first load, data is fetched and cached.
 - o Subsequent requests retrieve data from the cache until the cache expires.

Optional: Add Logging to See Caching in Action

Modify the controller to log whether data is retrieved from the cache or generated anew:

```
csharp

if (!_cache.TryGetValue(cacheKey, out products))
{
    Console.WriteLine("Cache miss. Fetching data...");
    // Fetch and cache data
}
else
{
}
```

```
        Console.WriteLine("Cache hit. Data retrieved from cache.");  
    }
```

Benefits of Caching in MVC:

1. Reduces database load by storing frequently accessed data.
2. Improves application performance by avoiding redundant computations.
3. Makes efficient use of server resources.

Documentation Reference:

- [ASP.NET Core Caching](#)



5. List and explain any five methods of IDistributedCache.



The **IDistributedCache** interface in ASP.NET Core provides methods to interact with distributed caching systems such as Redis, SQL Server, or NCache. Here are **five key methods** it offers and their explanations:

1. GetAsync(string key)

- **Description:** Asynchronously retrieves a cached value by its key.
- **Use Case:** When you need to fetch data stored in a distributed cache system.
- **Example:**

```
csharp

byte[] cachedData = await distributedCache.GetAsync("myKey");
```

2. SetAsync(string key, byte[] value, DistributedCacheEntryOptions options)

- **Description:** Asynchronously stores a value in the cache with specific expiration settings.
- **Use Case:** Useful for adding new items to the cache or updating existing entries with specified time-to-live (TTL).
- **Example:**

```
csharp

var cacheOptions = new DistributedCacheEntryOptions
{
    AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(10)
};
await distributedCache.SetAsync("myKey", myData, cacheOptions);
```

3. RemoveAsync(string key)

- **Description:** Asynchronously removes a cached entry by its key.
- **Use Case:** When an item is no longer needed or the cache should be cleared for the key.
- **Example:**

```
csharp

await distributedCache.RemoveAsync("myKey");
```


4. RefreshAsync(string key)

- **Description:** Asynchronously refreshes an entry's expiration time to prevent it from being removed due to expiration.
- **Use Case:** Ideal when you want to keep frequently accessed items in the cache for longer periods without resetting their values.
- **Example:**

```
csharp

await distributedCache.RefreshAsync("myKey");
```

5. Get(string key)

- **Description:** Synchronously retrieves a cached value by its key.
- **Use Case:** Similar to `GetAsync`, but for use in non-async contexts.
- **Example:**

```
csharp

byte[] cachedData = distributedCache.Get("myKey");
```

Key Points:

1. All methods that interact with the cache are available in both synchronous (`Get`, `Set`, `Remove`) and asynchronous (`GetAsync`, `SetAsync`, `RemoveAsync`) versions.
2. Expiration policies like **absolute expiration** and **sliding expiration** are handled through `DistributedCacheEntryOptions`.

Documentation Reference:

- [IDistributedCache Interface](#) (Microsoft Docs).



6. With respect to distributed cache, explain “AbsoluteExpiration”, “AbsoluteExpirationRelativeToNow” and “SlidingExpiration”.



In distributed caching, **expiration policies** help manage cached data by defining how long it should remain in the cache. Here’s an explanation of the three key expiration options:

1. AbsoluteExpiration

- **Definition:** Specifies a fixed point in time when the cached data will expire.
- **Behavior:** Once the specified time is reached, the cached item is removed regardless of whether it was accessed.
- **Use Case:** Ideal when you know the exact lifetime of the data and want it removed after a fixed period.
- **Example:**

```
csharp

var cacheOptions = new DistributedCacheEntryOptions
{
    AbsoluteExpiration = DateTimeOffset.UtcNow.AddMinutes(10)
};
```

- In this example, the data expires 10 minutes from now.

2. AbsoluteExpirationRelativeToNow

- **Definition:** Sets the expiration time relative to the current time. It defines how long the cached data should remain valid from the time it is added.
- **Behavior:** The countdown starts when the item is cached, and it expires after the specified duration.
- **Use Case:** Suitable when you want the data to stay in the cache for a specific period, regardless of access.
- **Example:**

```
csharp

var cacheOptions = new DistributedCacheEntryOptions
{
    AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(10)
};
```

- In this case, the item will expire 10 minutes after being cached.

3. SlidingExpiration

- **Definition:** Specifies that the expiration time should reset every time the cached item is accessed. The item remains in the cache as long as it is being used.
- **Behavior:** If the item is not accessed within the specified time, it expires.
- **Use Case:** Best for data that needs to stay in the cache as long as it is actively used, such as session data.
- **Example:**

```
csharp

var cacheOptions = new DistributedCacheEntryOptions
{
    SlidingExpiration = TimeSpan.FromMinutes(5)
};
```

- Here, the item will expire if it is not accessed for 5 minutes.

Comparison

Option	Lifetime Definition	Key Feature
AbsoluteExpiration	Fixed date/time	Hard expiry, regardless of use
AbsoluteExpirationRelativeToNow	Duration from current time	Relative lifetime
SlidingExpiration	Time since last access	Resets on every access

Notes:

1. You can combine **SlidingExpiration** with **AbsoluteExpiration** or **AbsoluteExpirationRelativeToNow** to enforce both policies.
2. Choose the appropriate policy based on the application requirements to optimize cache utilization.

For more details, check:

- [Microsoft Docs on Distributed Cache](#).



7. Write necessary code snippet and explain any two properties of MemoryCacheOptions.



Code Snippet for MemoryCacheOptions

csharp

```
using Microsoft.Extensions.Caching.Memory;
using System;

class Program
{
    static void Main()
    {
        // Create MemoryCacheOptions with custom properties
        var options = new MemoryCacheOptions
        {
            SizeLimit = 1024, // Limit cache size to 1024 units
            ExpirationScanFrequency = TimeSpan.FromMinutes(5) // Set scan frequency for expired items
        };

        // Create memory cache instance
        var memoryCache = new MemoryCache(options);

        // Add an item to the cache
        var cacheEntryOptions = new MemoryCacheEntryOptions
        {
            Size = 1, // Each cache item contributes to size limit
            SlidingExpiration = TimeSpan.FromMinutes(2)
        };
        memoryCache.Set("key", "cached data", cacheEntryOptions);

        // Retrieve item
        if (memoryCache.TryGetValue("key", out string cachedValue))
        {
            Console.WriteLine($"Cached Value: {cachedValue}");
        }
    }
}
```

Explanation of Two Properties of `MemoryCacheOptions`

1. `SizeLimit`:

- **Definition:** Specifies the maximum size limit of the cache in arbitrary units. When the cache size exceeds this limit, older items are evicted.
- **Use Case:** Helps control memory usage by limiting the total size of cached items.
- **Example in Code:**

csharp

```
var options = new MemoryCacheOptions
{
```

```
SizeLimit = 1024 // Cache size is limited to 1024 units
};
```

2. `ExpirationScanFrequency`:

- **Definition:** Determines how frequently the cache scans for expired items and removes them.
- **Use Case:** Ensures expired items are cleaned up efficiently without excessive performance overhead.
- **Example in Code:**

```
csharp

var options = new MemoryCacheOptions
{
    ExpirationScanFrequency = TimeSpan.FromMinutes(5) // Scan for expired items ev
    ery 5 minutes
};
```

Additional Notes:

- `SizeLimit` is only effective if you set the `Size` property for cache entries using `MemoryCacheEntryOptions`.
- Combine `ExpirationScanFrequency` with expiration policies (e.g., sliding or absolute) to manage cached data efficiently.

For more details, refer to [Microsoft Documentation](#).



8. Step-by-step explain how to use DistributedSqlServerCache.



Using **DistributedSqlServerCache** in an ASP.NET Core application involves setting up a SQL Server database to store cache entries and configuring the application to use it. Here's a step-by-step guide:

Step 1: Install Required NuGet Packages

1. Install the **Microsoft.Extensions.Caching.SqlServer** package:

```
bash

dotnet add package Microsoft.Extensions.Caching.SqlServer
```

Or, using the NuGet Package Manager:

- Search for `Microsoft.Extensions.Caching.SqlServer` and install it.
-

Step 2: Configure the SQL Server Cache Database

1. Use the **sql-cache** tool to create the SQL table required for caching.

```
bash

dotnet tool install --global dotnet-sql-cache
```

2. Run the following command to create a table named `CacheTable` in your SQL Server database:

```
bash

dotnet sql-cache create "Server=<YourServer>;Database=<YourDatabase>;Trusted_Connection
=True;" dbo CacheTable
```

Replace `<YourServer>` and `<YourDatabase>` with your actual SQL Server instance and database name.

This command creates a table with the following structure:

- **Id**: Unique identifier for each cache entry.
 - **Value**: Serialized cache data.
 - **ExpiresAtTime**: Expiration time for the entry.
 - **SlidingExpirationInSeconds**: Sliding expiration time (optional).
 - **AbsoluteExpiration**: Absolute expiration time (optional).
-

Step 3: Configure Services in `Startup.cs` or `Program.cs`

1. Add the `IDistributedCache` implementation for SQL Server in the `ConfigureServices` method:

```
csharp

using Microsoft.Extensions.DependencyInjection;

public void ConfigureServices(IServiceCollection services)
{
    services.AddDistributedSqlServerCache(options =>
    {
        options.ConnectionString = "Server=<YourServer>;Database=<YourDatabase>;Trusted
_Connection=True;";
        options.SchemaName = "dbo";
        options.TableName = "CacheTable";
    });

    services.AddControllersWithViews();
}
```

Step 4: Use the `IDistributedCache` in a Controller

1. Inject `IDistributedCache` into a controller:

```
csharp

using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Caching.Distributed;
using System.Text;

public class HomeController : Controller
{
    private readonly IDistributedCache _distributedCache;

    public HomeController(IDistributedCache distributedCache)
    {
        _distributedCache = distributedCache;
    }

    public async Task<IActionResult> Index()
    {
        string cacheKey = "SampleKey";
        string cachedValue = await _distributedCache.GetStringAsync(cacheKey);

        if (cachedValue == null)
        {
            // Cache miss: Set new value
            cachedValue = "Hello, Distributed SQL Server Cache!";
            var cacheOptions = new DistributedCacheEntryOptions
            {
                AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(10)
            };
            await _distributedCache.SetStringAsync(cacheKey, cachedValue, cacheOptions);
        }

        ViewBag.Message = cachedValue;
        return View();
    }
}
```

2. Create a `Views/Home/Index.cshtml` to display the cached value:

```
html
```

```
<h1>Distributed Cache Example</h1>
<p>@ViewBag.Message</p>
```

Step 5: Run the Application

1. Start the application and navigate to the appropriate route (e.g., `/Home/Index`).
2. Observe the caching behavior:
 - On the first request, the data is stored in the SQL Server cache.
 - On subsequent requests, data is retrieved from the cache instead of being recreated.

Key Benefits of DistributedSqlServerCache

1. Centralized caching system accessible across multiple servers.
2. Reliable storage with persistence (data remains even if the application restarts).
3. Suitable for web farm or cloud-based deployments.

Documentation

- Microsoft Docs: [Distributed SQL Server Cache](#).



9. List and explain any four properties of `SqlServerCacheOptions`.



Here are four key properties of `SqlServerCacheOptions` in ASP.NET Core, which are used to configure `DistributedSqlServerCache`:

1. ConnectionString

- **Description:** Specifies the connection string to connect to the SQL Server database where cache data is stored.
- **Purpose:** This is required to establish a connection with the database.
- **Example:**

```
csharp

options.ConnectionString = "Server=MyServer;Database=MyDatabase;Trusted_Connection=True;";
```

- **Use Case:** You use this property to define the database and server where your cache table resides.
-

2. SchemaName

- **Description:** Specifies the name of the database schema where the cache table is located.
- **Default Value:** ``dbo``
- **Purpose:** To organize the cache table within a specific schema in the database.
- **Example:**

```
csharp

options.SchemaName = "CacheSchema";
```

- **Use Case:** Useful when separating cache tables into their own schema for better organization or when using multiple applications with shared databases.
-

3. TableName

- **Description:** Specifies the name of the table where cache entries will be stored.
- **Default Value:** ``Cache``
- **Purpose:** To define or customize the table name used for caching.
- **Example:**

```
csharp
```

```
options.TableName = "AppCacheTable";
```

- **Use Case:** When you want to use a custom table name for better clarity or to avoid conflicts with existing database tables.

4. DefaultSlidingExpiration

- **Description:** Specifies the default sliding expiration time for cache entries if no expiration is explicitly set.
- **Default Value:** `TimeSpan.Zero`` (no sliding expiration).
- **Purpose:** Provides a fallback expiration mechanism for cached items.
- **Example:**

```
csharp
```

```
options.DefaultSlidingExpiration = TimeSpan.FromMinutes(30);
```

- **Use Case:** Useful when you want cache items to expire after a period of inactivity without setting expiration for every individual entry.

Code Example: Setting SqlServerCacheOptions

```
csharp
```

```
services.AddDistributedSqlServerCache(options =>
{
    options.ConnectionString = "Server=MyServer;Database=MyDatabase;Trusted_Connection=True;";
    options.SchemaName = "dbo";
    options.TableName = "CacheTable";
    options.DefaultSlidingExpiration = TimeSpan.FromMinutes(30);
});
```

References

1. [Microsoft Docs: Distributed SQL Server Cache.](#)
2. [SqlServerCacheOptions Class.](#)



10. What is response caching in ASP.NET Core MVC? When to use response caching? Write code snippet and explain response caching configuration.



What is Response Caching in ASP.NET Core MVC?

Response caching in ASP.NET Core MVC allows the server to cache HTTP responses to improve application performance and reduce the need to reprocess requests. It can cache responses either on the **server** or **client** (via cache control headers).

Key features include:

- Reducing the load on the server.
- Faster response times for repeated requests.
- Implementation based on HTTP caching standards.

When to Use Response Caching?

Use response caching when:

1. **Static or infrequently updated data** is requested, such as product details or blog pages.
2. Responses are **identical for multiple users** (public caching).
3. You want to improve the **performance of GET requests** by caching them client-side or on intermediate proxies.

Avoid response caching for:

- **Dynamic or user-specific content** like dashboards.
- Requests requiring **real-time updates**.

How to Configure Response Caching?

Step 1: Enable Response Caching Middleware

1. Add the middleware in `Program.cs`` or `Startup.cs``:

```
csharp

using Microsoft.AspNetCore.Builder;

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

// Enable response caching middleware
app.UseResponseCaching();

app.MapControllers();
app.Run();
```

Step 2: Configure Response Caching in Controllers

1. Use the `[ResponseCache]` attribute to control caching behavior at the action level:

```
csharp

using Microsoft.AspNetCore.Mvc;

[ApiController]
[Route("[controller]")]
public class ProductsController : ControllerBase
{
    [HttpGet("details")]
    [ResponseCache(Duration = 60, Location = ResponseCacheLocation.Any, NoStore = false)]
    public IActionResult GetProductDetails()
    {
        return Ok(new { ProductId = 1, Name = "Product A", Price = 99.99 });
    }
}
```

Explanation of Key Parameters

1. `Duration`: Specifies the cache duration in seconds.
Example: `Duration = 60` caches the response for 60 seconds.
2. `Location`: Defines where the cache is stored:
 - `Any`: Caching is allowed on client, server, or proxy.
 - `Client`: Cache is stored only on the client.
 - `None`: Caching is disabled.
3. `NoStore`: Indicates whether the cache should store the response.
 - `false` (default): The response can be stored.
 - `true`: Prevents caching.

Step 3: Add Cache Control Headers (Optional)

Use response headers to control caching manually when needed:

```
csharp

Response.Headers["Cache-Control"] = "public, max-age=60";
Response.Headers["Pragma"] = "no-cache"; // Optional
```

Full Example with Configuration

```
csharp

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

```
// Add services
builder.Services.AddControllers();

var app = builder.Build();

// Enable response caching middleware
app.UseResponseCaching();

app.MapGet("/example", (HttpContext context) =>
{
    context.Response.Headers["Cache-Control"] = "public, max-age=120";
    return Results.Ok("This is a cached response.");
});

app.Run();
```

References

1. [Response Caching Middleware - Microsoft Docs](#)
2. [ResponseCache Attribute in ASP.NET Core](#)



11. What are “Entity Framework Core” and “Object Relational Mapping”? Explain.



What is Entity Framework Core?

Entity Framework Core (EF Core) is a lightweight, extensible, and cross-platform **Object Relational Mapper (ORM)** for .NET applications. It allows developers to interact with databases using .NET objects, eliminating the need for most raw SQL code. EF Core is a modernized version of the original Entity Framework and supports LINQ (Language Integrated Query) to query the database.

Key Features of EF Core

1. **Cross-platform:** Works on Windows, macOS, and Linux.
2. **Code-First and Database-First:** Supports both workflows to define database schemas.
3. **Migration Support:** Helps manage database schema changes during development.
4. **Change Tracking:** Automatically detects changes in entities and saves them to the database.
5. **Query Support:** Leverages LINQ to write strongly-typed queries.
6. **Multiple Database Providers:** Supports SQL Server, SQLite, PostgreSQL, MySQL, and others.

Example

Instead of writing raw SQL to query a database:

```
sql

SELECT * FROM Products WHERE Price > 100
```

You can use EF Core with LINQ:

```
csharp

var expensiveProducts = dbContext.Products.Where(p => p.Price > 100).ToList();
```

What is Object Relational Mapping (ORM)?

Object Relational Mapping (ORM) is a technique for converting data between incompatible systems, specifically between **object-oriented programming languages** (like C# or Java) and **relational databases** (like SQL Server, MySQL, etc.). ORMs allow developers to work with database data as native objects instead of tables, rows, or SQL queries.

How ORM Works

1. Maps **classes** to database **tables**.
2. Maps **object properties** to **columns** in a table.
3. Maps **relationships** between objects (e.g., one-to-many, many-to-many) to relationships in the database.

Advantages of ORM

1. **Productivity**: Eliminates the need to write repetitive SQL queries.
2. **Maintainability**: Strongly-typed queries reduce runtime errors.
3. **Abstraction**: Developers work with objects rather than database tables.
4. **Cross-database Compatibility**: Allows switching between database providers without major code changes.

Example of ORM in EF Core

With an ORM like EF Core:

1. A class `Product`:

```
csharp

public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

2. Automatically maps to a database table `Products`:

```
sql

CREATE TABLE Products (
    Id INT PRIMARY KEY,
    Name NVARCHAR(100),
    Price DECIMAL
);
```

Summary

- **Entity Framework Core** is a modern ORM for .NET Core applications, allowing seamless integration with databases using object-oriented code.
- **Object Relational Mapping** bridges the gap between object-oriented programming and relational databases, improving development speed and reducing complexity.

For more details:

- [EF Core Documentation](#)
- [Object-Relational Mapping Overview](#)



12. Explain any three advantages and any two limitations of object relational mapping.



Advantages of Object Relational Mapping (ORM)

1. Improved Developer Productivity

ORMs like Entity Framework automate repetitive tasks like CRUD (Create, Read, Update, Delete) operations, reducing the need for writing raw SQL queries. This allows developers to focus on business logic instead of database interaction.

- Example: A single ORM method like ``dbContext.Users.Add(user)`` can replace multiple lines of SQL for inserting records.

2. Type Safety and Strongly-Typed Queries

ORMs enable developers to use strongly-typed objects and LINQ for queries, which are checked at compile-time, reducing runtime errors. This leads to more maintainable and less error-prone code.

- Example: A typo in ``dbContext.Users.FirstOrDefault(u => u.Username == "Admin")`` will be caught during compilation, unlike in a raw SQL query.

3. Database Independence

ORMs abstract database-specific queries, making it easier to switch between databases like SQL Server, MySQL, or PostgreSQL by changing the provider, without rewriting the codebase.

- Example: An application using EF Core can easily switch from SQL Server to SQLite with minor configuration changes.

Limitations of Object Relational Mapping (ORM)

1. Performance Overhead

ORMs can introduce performance issues for complex queries or large datasets, as the generated SQL may not be as optimized as handwritten queries.

- Example: An ORM might generate inefficient SQL for a query involving multiple joins or aggregations, requiring optimization through raw SQL or stored procedures.

2. Learning Curve and Complexity

ORMs have their own abstractions, APIs, and conventions that developers must learn. Misusing features like lazy loading or improper configurations can lead to unexpected behavior or performance issues.

- Example: Improper use of lazy loading can cause "N+1 query" problems, where multiple queries are sent to the database instead of a single optimized query.

References

1. [Microsoft Docs: EF Core Overview](#)
2. [Advantages and Disadvantages of ORM](#)
3. [Medium: Understanding ORM Pros and Cons](#)



13. Explain any five features of EF Core.



Here are five key features of **Entity Framework Core (EF Core)**:

1. Cross-Platform Support

EF Core is designed to work across multiple platforms, including Windows, macOS, and Linux. It supports .NET Core and .NET Framework, allowing developers to build and deploy applications in diverse environments.

- **Use Case:** Building cloud-based, containerized, or cross-platform applications.
- **Example:** Using EF Core in a .NET Core web API deployed on Linux.

2. Code-First and Database-First Approaches

EF Core supports both **Code-First** and **Database-First** development models:

- **Code-First:** Developers define the database schema using C# classes, and EF Core generates the database.
- **Database-First:** The database schema is defined first, and EF Core generates entity classes and relationships.
- **Example:**
 - Code-First:

csharp

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

Generates a `Products` table in the database.

3. LINQ Support

EF Core integrates with **Language Integrated Query (LINQ)** to enable querying databases using strongly-typed syntax in C#. This enhances productivity and reduces errors as queries are checked at compile time.

- **Example:**

```
csharp
```

```
var products = dbContext.Products.Where(p => p.Price > 100).ToList();
```

- **Advantage:** Eliminates the need for writing raw SQL queries.

4. Migrations

EF Core provides **database migrations**, which allow developers to incrementally update the database schema as the application evolves without losing existing data. Migrations are tracked in code, making schema updates version-controlled and maintainable.

- **Example:**

```
bash
```

```
dotnet ef migrations add InitialCreate
dotnet ef database update
```

- **Benefit:** Simplifies database schema management during development.

5. Multiple Database Providers

EF Core supports various database systems, making it versatile for different project requirements. Providers include:

- Microsoft SQL Server
- SQLite
- PostgreSQL
- MySQL
- In-Memory Database (for testing)
- **Example:** Switching between SQL Server and SQLite by changing the database provider in `DbContext`` configuration.

References

1. [EF Core Overview - Microsoft Docs](#)
2. [Entity Framework Core Features - GeeksforGeeks](#)
3. [EF Core Providers - Microsoft Docs](#)



14. What are "Code First" and "DB First" with respect to EF Core? Explain.



In the context of **Entity Framework Core (EF Core)**, **Code First** and **Database First** are two approaches for interacting with and managing databases in an application.

1. Code First Approach

Definition:

In the **Code First** approach, the database schema is generated from the application's code. Developers write **C# classes** (called entities) to define the data model, and EF Core translates these classes into database tables.

Key Features

- **Schema Generation:** EF Core creates the database schema based on entity classes and relationships.
- **Migrations:** Schema changes are handled using migrations, which incrementally apply updates to the database.
- **Control:** Gives developers complete control over the data model and schema evolution.

Steps in Code First

1. Define entity classes:

```
csharp

public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

2. Configure `DbContext`:

```
csharp

public class AppDbContext : DbContext
{
    public DbSet<Product> Products { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer("YourConnectionString");
    }
}
```

```
}  
}
```

3. Run migrations:

```
bash  
  
dotnet ef migrations add InitialCreate  
dotnet ef database update
```

Use Cases

- Greenfield projects where no existing database is present.
- Projects requiring frequent schema changes during development.

2. Database First Approach

Definition:

In the **Database First** approach, an existing database schema is used to generate entity classes and a ``DbContext``. EF Core reverse-engineers the database structure into code.

Key Features

- **Reverse Engineering:** Entity classes and relationships are created based on the database schema.
- **Customization:** The generated classes can be customized, though regeneration may overwrite changes.
- **Read-Only Databases:** Useful for working with legacy databases or when the schema is controlled externally.

Steps in Database First

1. Use EF Core tools to scaffold the database:

```
bash  
  
dotnet ef dbcontext scaffold "YourConnectionString" Microsoft.EntityFrameworkCore.SqlSe  
rver
```

2. EF Core generates:

- Entity classes for tables.
- A ``DbContext`` file for database interaction.

Use Cases

- Projects where a database already exists.

- Applications sharing a database with other systems.

Comparison of Code First and Database First

Feature	Code First	Database First
Definition	Database schema is generated from code.	Code is generated from the database schema.
Schema Changes	Controlled using migrations.	Limited; may require manual database updates.
Use Case	New projects, frequent schema evolution.	Legacy or shared databases.
Control	Full control over schema and code.	Limited; relies on existing schema.

Conclusion

- Choose **Code First** for new projects or when the data model evolves with the application.
- Use **Database First** for existing databases or systems where the schema is predefined or managed externally.

References

1. [EF Core Code First - Microsoft Docs](#)
2. [EF Core Database First - Microsoft Docs](#)
3. [EF Core: Code-First vs Database-First](#)



15. With respect to EF Core, write commands for the following:

- Uninstalling and installing Entity Framework Core Global Tool Package
- Testing the Entity Framework Core Global Tool
- Adding Entity Framework Core Package to a project



Here are the commands for managing and using Entity Framework Core Global Tools and Packages:

1. Uninstalling and Installing the Entity Framework Core Global Tool

Uninstall the Global Tool

To uninstall the EF Core global tool, use:

```
bash

dotnet tool uninstall --global dotnet-ef
```

Install the Global Tool

To install the EF Core global tool globally on your machine:

```
bash

dotnet tool install --global dotnet-ef
```

- Verify the installation:

```
bash

dotnet ef --version
```

2. Testing the Entity Framework Core Global Tool

To ensure that the EF Core global tool is correctly installed, run:

```
bash

dotnet ef
```

- This will display the available EF Core commands and options.
-

3. Adding Entity Framework Core Package to a Project

Basic EF Core Package

For adding EF Core to a project:

```
bash

dotnet add package Microsoft.EntityFrameworkCore
```

SQL Server Provider

To add EF Core with SQL Server support:

```
bash

dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

SQLite Provider

For SQLite support:

```
bash

dotnet add package Microsoft.EntityFrameworkCore.Sqlite
```

Tools for Design-Time Operations

To add tools for migrations and scaffolding:

```
bash

dotnet add package Microsoft.EntityFrameworkCore.Tools
```

References

- [EF Core Tools Documentation](#)
- [Package Manager Commands for EF Core](#)



16. Create C# context class and write code for AddDbContext with respect to EF Core. Also explain your solution.



C# Context Class and AddDbContext Code

Step 1: Create a Context Class

The context class in EF Core is derived from the `DbContext` class and is used to manage database interactions. It includes `DbSet` properties that represent tables in the database.

csharp

```
using Microsoft.EntityFrameworkCore;

public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }

    // Define DbSet properties for your entities (tables)
    public DbSet<Product> Products { get; set; }
    public DbSet<Customer> Customers { get; set; }

    // Optional: Configure relationships and other options using Fluent API
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        // Example: Configuring Product entity
        modelBuilder.Entity<Product>().HasKey(p => p.Id);
    }
}

// Sample entity classes
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}

public class Customer
{
    public int Id { get; set; }
    public string FullName { get; set; }
    public string Email { get; set; }
}
```

Step 2: Register the Context with Dependency Injection

The `AddDbContext` method is used to register the context with the Dependency Injection (DI) container in an ASP.NET Core application. This allows the application to manage the `DbContext` lifecycle and inject it where needed.

Add the following code in the `Startup.cs` file or in the `Program.cs` file (for .NET 6+).

csharp

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

// Add other services
builder.Services.AddControllers();

var app = builder.Build();

// Configure the HTTP request pipeline
app.UseAuthorization();
app.MapControllers();
app.Run();
```

Explanation

1. DbContext Class:

- `AppDbContext` inherits from `DbContext`, and it is the gateway for interacting with the database.
- The `DbSet` properties (`Products`, `Customers`) represent database tables.

2. Dependency Injection:

- The `AddDbContext` method registers the `AppDbContext` with the DI container.
- It configures the database provider (`UseSqlServer`) and connection string for database connectivity.

3. Connection String:

- The connection string is fetched from the `appsettings.json` file:

```
json

{
  "ConnectionStrings": {
    "DefaultConnection": "Server=YourServerName;Database=YourDatabaseName;Trusted_
Connection=True;"
  }
}
```

4. Lifecycle Management:

- EF Core ensures the `DbContext` instance is scoped to each HTTP request.

Usage Example

You can now inject `AppDbContext` into controllers or services:

csharp

```
public class ProductsController : ControllerBase
{
    private readonly AppDbContext _context;

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

    [HttpGet]
    public IActionResult GetAllProducts()
    {
        var products = _context.Products.ToList();
        return Ok(products);
    }
}
```

References

1. [Microsoft Docs: DbContext Class](#)
2. [EF Core Dependency Injection](#)
3. [ASP.NET Core with EF Core](#)



17. Step-by-step explain creating, applying migration and seeding the database.



Step-by-Step Guide: Creating, Applying Migrations, and Seeding the Database in EF Core

1. Setting Up Your Application

Before starting with migrations and seeding, ensure that your project has EF Core installed and configured with a context class and connection string.

Install EF Core and Tools

```
bash

dotnet add package Microsoft.EntityFrameworkCore
dotnet add package Microsoft.EntityFrameworkCore.Tools
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

2. Creating a Migration

Migrations are used to generate code that represents changes to the database schema.

Step 1: Add Migration

Run the following command in the terminal:

```
bash

dotnet ef migrations add InitialCreate
```

- This creates a folder named **Migrations** with a migration file, which contains the schema changes.

Example Migration File

```
csharp

protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.CreateTable(
        name: "Products",
        columns: table => new
        {
            Id = table.Column<int>(nullable: false)
```

```

        .Annotation("SqlServer:Identity", "1, 1"),
        Name = table.Column<string>(nullable: true),
        Price = table.Column<decimal>(nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Products", x => x.Id);
    });
}

protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropTable(name: "Products");
}

```

3. Applying Migrations to Update the Database

Use the `dotnet ef database update` command to apply the migrations to the database.

Command

```

bash

dotnet ef database update

```

- This updates the database schema to match the current migration.

4. Seeding the Database

Seeding is used to populate the database with initial data when migrations are applied.

Step 1: Configure Seeding in the Context Class

Use the `ModelBuilder` in the `OnModelCreating` method of your `DbContext`.

```

csharp

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    // Seed data for the Products table
    modelBuilder.Entity<Product>().HasData(
        new Product { Id = 1, Name = "Laptop", Price = 1200 },
        new Product { Id = 2, Name = "Smartphone", Price = 800 }
    );
}

```

Step 2: Update Migrations

After adding seed data, create and apply a new migration.

```
bash
```

```
dotnet ef migrations add SeedData
dotnet ef database update
```

5. Verify the Database

Use a database tool like **SQL Server Management Studio** or **Azure Data Studio** to check that:

- The database schema is created as per the migrations.
- The seeded data is added to the respective tables.

6. Summary of Commands

Command	Purpose
<code>`dotnet ef migrations add <Name>`</code>	Creates a new migration.
<code>`dotnet ef database update`</code>	Applies pending migrations to the database.
<code>`dotnet ef migrations remove`</code>	Removes the last migration.

When to Use These Steps

- Use migrations for schema evolution during development.
- Use seeding to add default data, such as admin users or configuration settings.

References

1. [EF Core Migrations Documentation](#)
2. [Seeding Data in EF Core](#)
3. [EF Core Getting Started](#)



4. Explain the following code snippet:

```
public IActionResult Index()
{
    string? totalString;
    bool isExist = memoryCache.TryGetValue("result1", out totalString);
    if (!isExist)
    {
        int count = 1000000000;
        long total = 0;
        for (int i = 1; i <= count; i++)
        {
            total += i;
        }
        totalString = $"{DateTime.Now.ToLongTimeString()} {total}";

        var cacheEntryOptions = new MemoryCacheEntryOptions()
            .SetSlidingExpiration(TimeSpan.FromSeconds(30));

        memoryCache.Set("result1", totalString, cacheEntryOptions);
    }

    return View("Index", totalString);
}
```



Explanation of the Code Snippet

This code demonstrates how to use **in-memory caching** in an ASP.NET Core MVC application to store and retrieve a computed value efficiently. Let's break it down step-by-step:

Code Analysis

1. Declaration and Check Cache:

csharp

```
string? totalString;
bool isExist = memoryCache.TryGetValue("result1", out totalString);
```

- **Purpose:** Checks if a cached value with the key `"result1"` exists in the memory cache.
- **Outcome:**
 - If it exists (`isExist = true`), the cached value is retrieved into `totalString`.
 - If it doesn't exist (`isExist = false`), the block of code inside the `if` condition is executed to compute and store the value.

2. Compute Value (if Not in Cache):

csharp

```
int count = 1000000000;
long total = 0;
for (int i = 1; i <= count; i++)
{
    total += i;
}
totalString = $"{DateTime.Now.ToLongTimeString()} {total}";
```

- **Purpose:** Computes the sum of integers from `1` to `1,000,000,000` (a computationally expensive operation).
- **Additional Info:** The computed value is concatenated with the current time (e.g., `"10:30:00 500000000500000000"`).

3. Set Cache Entry:

csharp

```
var cacheEntryOptions = new MemoryCacheEntryOptions()
    .SetSlidingExpiration(TimeSpan.FromSeconds(30));

memoryCache.Set("result1", totalString, cacheEntryOptions);
```

- **Purpose:** Stores the computed value in the memory cache with the key `"result1"`.
- **Sliding Expiration:** Ensures that the cached value remains in the cache for **30 seconds of inactivity**. Each access resets the expiration timer.

4. Return Cached/Computed Value:

csharp

```
return View("Index", totalString);
```

- **Purpose:** Passes the cached or computed value (`totalString`) to the `Index` view for display.

Key Concepts Demonstrated

1. Caching Mechanism:

- Reduces redundant computation by storing expensive results in memory.
- Results are fetched from the cache on subsequent requests within the expiration window.

2. Sliding Expiration:

- Extends the lifetime of a cached value as long as it is accessed regularly.

3. Efficient Use of Resources:

- Reduces CPU-intensive operations by serving precomputed results from memory.

Use Case

This technique is suitable for scenarios where:

- The result of a computation is costly to calculate.
 - The result does not change frequently.
 - Performance optimization is a priority.
-

Potential Improvements

1. **Cache Key Naming:** Use meaningful, unique keys to avoid collisions.
 2. **Error Handling:** Add error handling for potential exceptions in computation or caching.
 3. **Dynamic Expiration:** Adjust expiration time based on application needs.
-

Output Example

On the first request, the sum is computed, and the current time (e.g., `10:30:00`) is appended. Subsequent requests within 30 seconds return the same cached value without recomputation.

References

1. [MemoryCache in ASP.NET Core](#)
2. [Sliding Expiration in Caching](#)