.NET Microservices – Azure DevOps and AKS

Section 9: Caching - Notes

Introduction to Caching

Caching is a process of temporarily storing frequently accessed data in memory, allowing faster retrieval in future requests. This reduces the need to fetch data from the primary data source (like databases) for every request. Caching plays a crucial role in enhancing the performance and scalability of applications, particularly in high-traffic environments, by reducing latency and lowering the load on databases and APIs.

Types of Caching

1. **In-Memory Caching**:
    - Stores data directly in the memory (RAM) of the local application server.
    - Examples: ASP.NET Core's **MemoryCache**.
    - Suitable for small-scale applications with a single server, but doesn't work well in **distributed systems**.
2. **Distributed Caching**:
    - Stores cache data in a centralized, shared location accessible to multiple servers.
    - It's crucial in **microservices** and **cloud-based architectures**, where applications are spread across many servers.
    - **Redis** is a popular choice for distributed caching.

In distributed systems, local in-memory caching may lead to inconsistencies because each server holds its own cache. A distributed cache ensures consistency across all instances of the application.

What is Redis?

Redis (Remote Dictionary Server) is an open-source, in-memory data structure store that can be used as a **cache**, database, or message broker. It is designed for high-performance operations like caching, as it stores everything in memory, making data retrieval incredibly fast. Redis supports complex data types such as strings, lists, sets, and hashes, making it versatile for many use cases.

Why Redis for Caching?

- **In-Memory**: Redis is an in-memory cache, meaning that all data is stored in RAM, ensuring very low latency in data retrieval.
- **High Performance**: Redis is capable of handling millions of requests per second due to its lightweight nature.
- **Persistence Options**: While Redis is primarily in-memory, it can persist data to disk if needed, combining speed with reliability.
- **Scalability**: Redis can operate in distributed environments, making it ideal for large-scale, cloud-based applications.
- **Data Structures**: Redis supports multiple data types (strings, lists, hashes, sets), which can cater to various caching requirements.

Example: An e-commerce site can cache user profiles, product information, and shopping cart details in Redis, which can be quickly accessed during peak traffic without needing a round trip to the database.

Key Redis Features for Caching

- **TTL (Time-To-Live)**: You can set expiration times for cached data, ensuring that stale data is automatically purged.
- **Eviction Policies**: Redis offers configurable eviction policies (e.g., Least Recently Used - LRU) to remove old or unused cache entries when memory limits are reached.
- **Cluster Support**: Redis can be scaled horizontally using clusters to distribute the load across multiple nodes.
- **Pub/Sub Messaging**: Redis can also be used for messaging between microservices through its publish-subscribe mechanism.

Common Use Cases of Redis Caching in ASP.NET Core

- **Session State Management**: Redis is used to store user session data in distributed systems.
- **Output Caching**: Cache the output of expensive computations or frequently accessed web pages, improving response times.
- **Database Query Results**: Cache results of complex database queries to reduce database load.
- **API Caching**: Cache API responses, reducing the need for repetitive external calls.

Example: If you have an expensive query fetching top 10 products from a database, caching the result in Redis can reduce database load. Instead of querying the database every time, the system checks if the result exists in Redis first.

Why Use Redis over Other Caching Solutions?

- **Scalability**: Redis handles distributed caching well, making it suitable for cloud and microservice-based architectures.
- **Data Durability (Optional)**: Redis can persist data to disk, ensuring durability beyond simple in-memory caching.
- **Wide Language Support**: Redis supports a wide variety of programming languages, including C#, Java, Python, and more.
- **Replication & High Availability**: Redis offers built-in replication, meaning data can be replicated across multiple servers to ensure high availability.

Redis is widely used in high-traffic environments like e-commerce platforms, content management systems, and IoT applications to ensure smooth and fast data retrieval.

Summary (for Interview Preparation)

- **What is Redis?**: A high-performance, in-memory data store used for caching, databases, and message brokering.
- **Benefits of Caching**: Faster data retrieval, reduced load on the primary data source, improved scalability, and performance.
- **Distributed Caching**: Redis enables distributed caching, making it suitable for cloud-based and microservice architectures.
- **Redis Key Features**: In-memory caching, data persistence options, TTL (Time-To-Live), eviction policies, support for complex data types, and clustering.
- **Use Cases**: Session management, output caching, database query results caching, API response caching, etc.

Redis NuGet Package

In ASP.NET Core, **Redis** is integrated into your application through the **StackExchange.Redis** NuGet package, a popular and high-performance client library used to communicate with Redis from .NET applications.

Step 1: Installing the Redis NuGet Package

To use Redis in your ASP.NET Core application, you need to install the **StackExchange.Redis** package. Here's how to do it:

1. **Via NuGet Package Manager**:
    - In Visual Studio, right-click on your project, select "Manage NuGet Packages."
    - Search for "StackExchange.Redis."
    - Select the package and install the latest stable version.
2. **Via .NET CLI**: You can also install the package using the .NET CLI by running the following command in your project directory:

dotnet add package StackExchange.Redis

This installs the **StackExchange.Redis** client, which provides easy-to-use methods for connecting to Redis and performing operations like setting and retrieving cache values.

Step 2: Configuring Redis in ASP.NET Core

After installing the NuGet package, the next step is to configure Redis as the distributed cache provider in your ASP.NET Core application.

You need to modify the Program.cs or Startup.cs file to register Redis as a service.

Here's how to set up Redis in **Program.cs**:

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


// Configure Redis as the distributed cache
builder.Services.AddStackExchangeRedisCache(options =>
{
   options.Configuration = builder.Configuration.GetConnectionString("RedisConnection");
   options.InstanceName = "RedisCacheInstance";
});
var app = builder.Build();
app.Run();
```

Explanation:

- AddStackExchangeRedisCache: This extension method configures Redis as the distributed cache provider.
- options.Configuration: This specifies the Redis server connection string (e.g., localhost:6379 for local testing, or the Redis server URL for production).
- options.InstanceName: A logical name used to prefix Redis keys, allowing multiple applications to share the same Redis instance without key collisions.

Connection String for Redis:

The Redis connection string is typically stored in the appsettings.json file:

```
{
  "ConnectionStrings": {
    "RedisConnection": "localhost:6379"
  }
}
```

- **localhost:6379** is the default address for a locally running Redis instance. Replace this with the Redis server's IP address or hostname when deploying to a production environment.

Step 3: Accessing the Distributed Cache

In ASP.NET Core, the **IDistributedCache** interface provides methods to interact with the distributed cache (which, in this case, will be Redis). The following methods are available in the IDistributedCache interface:

- **GetAsync(string key)**: Retrieves the cached value for the given key.
- **SetAsync(string key, byte[] value)**: Caches the value for the specified key.
- **RemoveAsync(string key)**: Removes the cached value associated with the given key.
- **RefreshAsync(string key)**: Extends the expiration of a cached item, keeping it alive.

These methods will be used to read from and write to the Redis cache.

Summary (for Interview Preparation)

- **StackExchange.Redis**: A high-performance client library used in .NET for communicating with Redis.
- **Installation**: Install the package via **NuGet** or **.NET CLI**.

- **Configuration**: Redis is registered as the distributed cache provider using AddStackExchangeRedisCache in the service configuration.
- **Connection String**: Typically configured in appsettings.json. For local Redis, use localhost:6379; for production, use the Redis server's IP or URL.
- **IDistributedCache** Interface: Provides methods like GetAsync, SetAsync, RemoveAsync, and RefreshAsync to work with Redis as a distributed cache.

Redis Docker Image

When developing or testing with Redis, you can easily run Redis as a Docker container instead of installing it directly on your system. Using Docker allows for quick setup and easy removal of the Redis instance when you're done.

Step 1: Pulling the Redis Docker Image

The official Redis Docker image is available on **Docker Hub**. You can pull it using the following command:

docker pull redis

This command downloads the latest official Redis image from Docker Hub, which is maintained by the Redis team.

Step 2: Running the Redis Container

After pulling the image, you can run a Redis container using the docker run command:

docker run --name redis-server -p 6379:6379 -d redis

- **--name redis-server**: This names the container redis-server for easy reference.
- **-p 6379:6379**: Maps port 6379 on your local machine to port 6379 inside the container (Redis runs on port 6379 by default).
- **-d redis**: Runs the Redis container in detached mode (in the background).

This starts a Redis container that will listen for requests on localhost:6379.

Step 3: Verifying Redis Is Running

To verify that the Redis container is running, use the following Docker command:

docker ps

This command lists all running containers. You should see an entry for the redis-server container, with a status indicating it is up and running.

Example output:

```
CONTAINER ID   IMAGE     COMMAND       CREATED         STATUS         PORTS                    NAMES
abcd1234       redis     "docker-entry..."   1 minute ago    Up 1 minute    0.0.0.0:6379->6379/tcp   redis-server
```

The output shows that Redis is mapped to port **6379** on your local machine.

Step 4: Interacting with Redis in the Container

You can execute Redis commands directly inside the running container by using the docker exec command.

To open the Redis CLI inside the container, run:

docker exec -it redis-server redis-cli

This opens an interactive terminal inside the Redis container. You can now run Redis commands, such as PING, to check connectivity:

PING

The expected response is:

PONG

This confirms that Redis is running and responding correctly inside the Docker container.

Step 5: Stopping and Removing the Redis Container

When you are done working with the Redis container, you can stop and remove it with the following commands:

1. **Stopping the container**:

docker stop redis-server

2. **Removing the container**:

docker rm redis-server

These commands will stop the Redis container and remove it from Docker, cleaning up your environment.

Summary (for Interview Preparation)

- **Redis Docker Image**: Available on Docker Hub and can be pulled using docker pull redis.
- **Running the Redis Container**: Use docker run -p 6379:6379 -d redis to run Redis as a container, mapping it to port 6379.
- **Verifying Redis**: Use docker ps to check that the Redis container is running.
- **Interacting with Redis**: Use docker exec -it redis-server redis-cli to open the Redis CLI inside the container.
- **Stopping and Removing**: Use docker stop and docker rm to stop and remove the Redis container when it's no longer needed.

Reading and Writing into Redis Cache (using IDistributedCache)

1. Introduction to IDistributedCache

In .NET Core, the IDistributedCache interface provides a mechanism for working with distributed caches like Redis. This interface allows you to perform caching operations in a unified way, regardless of the caching provider (Redis, SQL Server, etc.). When using Redis, IDistributedCache offers an abstraction to easily interact with Redis as a key-value store.

Key methods of IDistributedCache:

- **GetAsync**: Retrieves a cached item.
- **SetAsync**: Stores an item in the cache.
- **RemoveAsync**: Removes an item from the cache.
- **RefreshAsync**: Refreshes the expiration of a cache entry.

When working with Redis, these methods map to common Redis commands (e.g., GET, SET, DEL).

2. Configuring Redis in the .NET Core Application

To use Redis with IDistributedCache, we first need to configure Redis in the application's Program.cs or Startup.cs.

public void ConfigureServices(IServiceCollection services)

```
{
    // Add Redis to IDistributedCache
    services.AddStackExchangeRedisCache(options =>
    {
        options.Configuration = Configuration.GetConnectionString("RedisConnection");
        options.InstanceName = "MyRedisInstance"; // Optional, to namespace cache keys
    });
}
```

- **AddStackExchangeRedisCache**: This method registers Redis as the distributed cache provider in .NET Core using the StackExchange.Redis library.
- **options.Configuration**: This is the Redis connection string, typically including the Redis server's IP and port.
- **options.InstanceName**: Provides an optional namespace prefix for your cache keys, which can help in multi-application environments.

3. Service Class for Caching (Using IDistributedCache)

Let's now create a simple service class that uses IDistributedCache to perform caching operations. In this example, we'll create a ProductCacheService that reads from and writes to Redis.

Example:

```
using Microsoft.Extensions.Caching.Distributed;
using System.Text.Json;
using System.Threading.Tasks;

public class ProductCacheService
{
    private readonly IDistributedCache _cache;

    public ProductCacheService(IDistributedCache cache)
    {
        _cache = cache;
    }
```

```
    // Fetch product from cache by ID
    public async Task<ProductDTO> GetProductAsync(string productId)
    {
        var cachedProduct = await _cache.GetStringAsync(productId);

        if (cachedProduct != null)
        {
            return JsonSerializer.Deserialize<ProductDTO>(cachedProduct);
        }

        return null; // Product not found in cache
    }

    // Store product in cache with expiration
    public async Task SetProductAsync(string productId, ProductDTO product)
    {
        var cacheOptions = new DistributedCacheEntryOptions()
            .SetAbsoluteExpiration(TimeSpan.FromMinutes(30)); // Expiration time of 30 minutes

        var serializedProduct = JsonSerializer.Serialize(product);

        await _cache.SetStringAsync(productId, serializedProduct, cacheOptions);
    }

    // Remove product from cache
    public async Task RemoveProductAsync(string productId)
    {
        await _cache.RemoveAsync(productId);
    }
}
```

Explanation of Code:

1. **Constructor Injection**: We inject the IDistributedCache service into our ProductCacheService to interact with Redis.
2. **GetProductAsync**:
    - Uses GetStringAsync to retrieve a cached product by its ID.
    - The product data, if found, is deserialized from JSON back into a ProductDTO.
3. **SetProductAsync**:
    - Takes a ProductDTO and stores it in Redis using SetStringAsync.
    - Sets an absolute expiration of 30 minutes, meaning the cache entry will be removed after that time.
4. **RemoveProductAsync**:
    - Deletes the cached product from Redis using RemoveAsync.

4. Interacting with the Service

Here's an example of how you might interact with this service in a controller:

```
[ApiController]
[Route("api/[controller]")]
public class ProductsController : ControllerBase
{
    private readonly ProductCacheService _cacheService;

    public ProductsController(ProductCacheService cacheService)
    {
        _cacheService = cacheService;
    }

    [HttpGet("{id}")]
    public async Task<IActionResult> GetProduct(string id)
    {
        var product = await _cacheService.GetProductAsync(id);
```

```
        if (product != null)
        {
            return Ok(product); // Return cached product
        }

        // Simulate fetching product from database (if not cached)
        product = await GetProductFromDatabaseAsync(id);

        if (product != null)
        {
            await _cacheService.SetProductAsync(id, product); // Cache the product
            return Ok(product);
        }

        return NotFound(); // Product not found
    }

    private async Task<ProductDTO> GetProductFromDatabaseAsync(string productId)
    {
        // Simulated database fetch logic
        return new ProductDTO { ProductID = Guid.NewGuid(), ProductName = "Sample Product" };
    }
}
```

Key Points:

- **Caching in the Controller**: Before fetching from the database, we first try to retrieve the product from Redis. If the product is not found in the cache, it's fetched from the database, cached, and then returned to the client.
- **Cache Expiration**: We set a 30-minute expiration to avoid serving outdated data indefinitely.

Summary (for Interview Preparation)

- **IDistributedCache Interface**: A standard interface in .NET Core to interact with different caching providers like Redis, SQL Server, etc.
- **StackExchange.Redis**: The package used for Redis caching in .NET Core.
- **Basic Methods**: GetAsync, SetAsync, RemoveAsync are the main methods to interact with the cache.
- **Expiration**: Cache entries can have absolute or sliding expiration.
- **Service Layer Integration**: The service layer (like ProductCacheService) abstracts caching logic, making the controller code cleaner and more maintainable.