Resilience and Caching in .NET with C#

Resilience & Transient Faults

- **Transient faults** are brief, self-correcting failures (e.g. network blips, timeouts, overloaded servers). Retrying after a short delay can succeed.

- **Resilience** means designing apps to handle such faults gracefully, improving reliability and user experience. For example, a resilient service will retry a failed HTTP call or use a fallback value instead of crashing.

- **Polly** is the de facto .NET library for resilience/transient-fault handling. It provides configurable **policies** (Retry, Circuit Breaker, Bulkhead, Timeout, Fallback, etc.) that you can apply declaratively to your code.

Polly Features

- **Retry Policies**: Automatically retry failed operations with configurable count, backoff, and exception handling. Useful for short-term glitches.

- **Circuit Breaker**: Stops repeated attempts if a component is failing. Opens the circuit after a threshold of faults (e.g. *fail 5 times*), then rejects calls for a cooldown period. Prevents overloading failing services.

- **Bulkhead Isolation**: Limits concurrency to a resource, isolating faults. E.g. only 6 concurrent calls to an API, others queue or fail. This ensures one slow part doesn’t consume all threads.

- **Timeout Policies**: Fails operations that exceed a time limit. Use `Policy.TimeoutAsync(TimeSpan)` so hung calls are aborted quickly.

- **Fallback Policy**: Provides a default result or alternative action when all else fails. Acts as a last-resort to return a safe value or notify the user.

Retry Policy Example

Define a retry policy with exponential backoff. Example using Polly’s **WaitAndRetryAsync**:

```
 static IAsyncPolicy<HttpResponseMessage> GetRetryPolicy() {
    return HttpPolicyExtensions
        .HandleTransientHttpError()  // Handles 5XX, network failures
        .OrResult(msg => msg.StatusCode == HttpStatusCode.NotFound)
        .WaitAndRetryAsync(6, retryAttempt =>
            TimeSpan.FromSeconds(Math.Pow(2, retryAttempt))
        );
}
```

- This policy retries up to 6 times, doubling the delay each attempt.

Integration: With `IHttpClientFactory` (in ASP.NET Core), attach policies when configuring the client:

```
 builder.Services.AddHttpClient<IBasketService, BasketService>()
       .SetHandlerLifetime(TimeSpan.FromMinutes(5))
       .AddPolicyHandler(GetRetryPolicy())
       .AddPolicyHandler(GetCircuitBreakerPolicy());
```

- Here we add both retry and circuit-breaker policies to the HTTP client pipeline.

Circuit Breaker Example

Create a circuit-breaker policy:

```
 static IAsyncPolicy<HttpResponseMessage> GetCircuitBreakerPolicy() {
    return HttpPolicyExtensions
        .HandleTransientHttpError()
        .CircuitBreakerAsync(5, TimeSpan.FromSeconds(30));
}
```

- This opens the circuit for 30 seconds after 5 consecutive failures. Subsequent calls fail immediately until the period elapses, preventing further strain on the failing service.

Bulkhead Isolation Example

Define a bulkhead policy to limit concurrency:

```
 // Allow max 6 concurrent executions, unlimited queue
var bulkheadPolicy = Policy.BulkheadAsync<HttpResponseMessage>(6,
int.MaxValue);
```

- This ensures only 6 threads can execute the protected code at once. Excess requests will queue, avoiding resource exhaustion. Adjust the limits to your app’s capacity.

Timeout Policy Example

Use a timeout to fail long-running tasks:

```
 // Fail if not completed in 10 seconds
var timeoutPolicy =
Policy.TimeoutAsync<HttpResponseMessage>(TimeSpan.FromSeconds(10));
```

- If the HTTP call (or other operation) takes longer than 10 seconds, Polly will cancel it and throw a `TimeoutRejectedException`.

Fallback Policy Example

Provide a safe fallback result when all else fails:

```
 // Return a default response if the call fails
IAsyncPolicy<HttpResponseMessage> fallbackPolicy = Policy<HttpResponseMessage>
    .Handle<Exception>()
    .FallbackAsync(
        // Fallback action:
        Task.FromResult(new HttpResponseMessage(HttpStatusCode.OK) {
            Content = new StringContent("Default response")
        }),
        // OnFallback: logging or notification (optional)
        onFallbackAsync: async (delegateResult, context) => {
            Console.WriteLine("Fallback triggered: " + delegateResult.Exception?.Message);
        });
```

- This catches any exception and returns a default HTTP response. The `onFallbackAsync` handler can be used to log the failure.

Combining and Integrating Polly

Policy Wraps: You can combine multiple policies in a chain. For example, wrap retry then fallback:

```
 var policyWrap = Policy.WrapAsync(fallbackPolicy, retryPolicy);
await policyWrap.ExecuteAsync(() => httpClient.GetAsync("http://example.com"));
```

- Here, the request is retried first; if all retries fail, the fallback provides a default result.

- **Dependency Injection:** Register policies and handlers in `Program.cs` or `Startup.cs` with `AddHttpClient`, or use custom middleware. The `Microsoft.Extensions.Http.Polly` package provides tight integration.

- **Use Cases:** Apply Polly policies to external calls (HTTP, gRPC), database queries, or any I/O where transient faults occur. This makes your API or service resilient to momentary issues.

Caching Strategies Overview

- **Purpose:** Caching stores frequently-used data in fast storage to improve performance and scalability. Use caching for read-heavy, infrequently changing data.

- **Cache Types:**

  - *In-Memory Cache (`IMemoryCache`)* – stores objects in the server's memory. Very fast but local to one server (use sticky sessions or failover considerations).

  - *Distributed Cache (`IDistributedCache`)* – stores data externally (e.g. Redis, NCache, SQL Server) shared by all servers. Scales across instances and survives restarts.

In-Memory vs Distributed Cache

- **IMemoryCache:** Per-server cache of actual objects (live types). Ideal for small-scale apps or single-server scenarios. Requires sticky sessions if running in a web farm.

- **Distributed Cache:** Shared by multiple app instances. Data is usually stored as byte[] (serialized). Benefits: consistent data across servers, surviving restarts, and higher scale-out. Examples include Redis and NCache (an open-source .NET cache).

Example: Configure Redis in ASP.NET Core:

```
 builder.Services.AddStackExchangeRedisCache(options => {
   options.Configuration = builder.Configuration.GetConnectionString("Redis");
   options.InstanceName = "MyAppCache";
});
```

- After this, inject `IDistributedCache` to get/set values in Redis.

Caching Patterns

- **Cache-Aside (Lazy Loading):** Application first checks the cache. On **miss**, it loads data from the primary store, then adds it to the cache. On reads:

    1. Try `cache[key]`.

    2. If found, return it.

    3. If not, retrieve from DB, store in cache, then return.
       This ensures only requested data is cached. *(Drawback: initial request on a miss is slower).*

- **Write-Through:** Application writes data to the cache **and** the data store synchronously. The cache update happens immediately on data change. The cache always reflects the primary store. *(This adds some overhead on writes but simplifies consistency.)*

- **Write-Behind (Write-Back):** Application writes **only** to the cache first, and the cache layer asynchronously propagates changes to the database after a delay. This speeds up writes (no DB wait) at the cost of eventual consistency and risk of data loss if the cache fails before write-behind flush.

- **Eviction/Expiration:** Cached items can expire or be evicted to control memory. Configure **Absolute** expiration (item expires after fixed time) or **Sliding** expiration (expires if not accessed within time). The cache also evicts (e.g. LRU – least recently used) when exceeding memory limits. Always set sensible expirations to prevent stale data.

- **Consistency:** In write-through/behind and cache-aside, ensure invalidations on updates. For example, on a DB update, either update the cache (write-through) or remove the old value (cache-aside) so the next read gets fresh data.

In-Memory Cache Example (IMemoryCache)

Setup (ASP.NET Core):

```
 services.AddMemoryCache();
```

- Inject `IMemoryCache` into your controller/service.

Usage (Cache-Aside):

```
 private readonly IMemoryCache _cache;
public MyService(IMemoryCache cache) { _cache = cache; }

public async Task<MyData> GetDataAsync(string id) {
    // Try get from cache
    if (!_cache.TryGetValue(id, out MyData cached)) {
        // Cache miss: load from DB or API
        cached = await LoadFromDataStore(id);
        // Store in cache with expiration
        _cache.Set(id, cached, new MemoryCacheEntryOptions {
            AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(5)
        });
    }
    return cached;
}
```

- Here, on a miss the data is loaded and then `Set` in cache with a 5-minute TTL.

Distributed Cache Example (Redis)

Setup: (StackExchange.Redis)

```
 builder.Services.AddStackExchangeRedisCache(opt => {
    opt.Configuration = "localhost:6379";
    opt.InstanceName = "AppCache:";
});
```

Usage (Cache-Aside):

```
 private readonly IDistributedCache _cache;
public MyService(IDistributedCache cache) { _cache = cache; }

public async Task<string> GetDataAsync(string key) {
    // Try get from distributed cache (string values for simplicity)
    var cached = await _cache.GetStringAsync(key);
    if (cached != null) {
        return cached; // Cache hit
    }
    // Miss: retrieve from data source
    string data = await LoadFromDataStoreAsync(key);
    // Save to cache with expiration
    var options = new DistributedCacheEntryOptions
        .SetAbsoluteExpiration(TimeSpan.FromMinutes(10));
    await _cache.SetStringAsync(key, data, options);
    return data;
}
```

- This demonstrates cache-aside with Redis. You can similarly use `_cache.GetAsync`/`SetAsync` for byte[].

Caching Patterns in .NET Code

- **Cache-Aside (Lazy):** The above examples for MemoryCache and Redis illustrate cache-aside. The app explicitly manages cache population.

Write-Through: In .NET you could wrap your data layer so that on every write, you update both the DB and cache. E.g.:

```
 await UpdateDatabaseAsync(key, newValue);
_cache.SetString(key, newValue); // Write-through to cache
```

-
- **Write-Behind:** .NET doesn’t provide built-in write-behind. You’d implement it using background tasks or cache support (some distributed caches like Redis Enterprise support write-behind recipes). Typically, you asynchronously batch DB updates

Cache Expiration & Eviction

- **Expiration:** Always set cache TTLs (`AbsoluteExpiration` or `SlidingExpiration`) to avoid stale data and control size. For example, expiring session data after short intervals, and product data after longer.
- **Eviction:** In-memory caches use eviction (e.g. LRU) when memory is low. You can also use size-based limits (`SizeLimit` in IMemoryCache). Distributed caches like Redis also support eviction policies (e.g. `volatile-lru`).
- **Cache Refresh:** For critical data, consider refreshing before expiration (refresh-ahead) or on cache hit upon detecting stale info.

Further Resources and Libraries

- **Polly Documentation:** The Polly Project and Polly’s GitHub.
- **Microsoft Docs on Resilience:** .NET Microservices guides cover Retry/CircuitBreaker patterns.
- **Caching Docs:** Microsoft’s ASP.NET Core caching docs (IMemoryCache and IDistributedCache).
- **Libraries:** Redis (via `Microsoft.Extensions.Caching.StackExchangeRedis`), and **NCache** (open-source .NET cache by Alachisoft) are popular distributed cache solutions.