Singleton pattern in ASP.NET

Singleton means: *there is exactly one instance of a type for the lifetime of an application scope*, and everyone uses that same instance.

In **ASP.NET / ASP.NET Core**, there are **two common meanings** people mix up:

1. **Classic "GoF Singleton"**
   A class enforces "only one instance" via static fields + private constructor.
2. **ASP.NET Core DI Singleton (the one you usually want)**
   You register a service as **Singleton** in the dependency injection container, meaning:
   - **One instance per application process** (per host)
   - Shared across **all requests** and **all users**
   - Created once (either lazily on first use or at startup, depending on how you register/resolve)

In modern ASP.NET Core apps, you almost always prefer **DI-managed singletons** over manually-coded static singletons.

The problems it solves

Singleton is used when:

- Creating the object is expensive, and you want to reuse it
- You need a shared, consistent "application-level" service
- The service is **stateless** or properly **thread-safe**

Because in a web app, **many requests run concurrently**, a singleton must handle multi-threading correctly.

Singleton services must be:

- **Thread-safe**
- Usually **stateless**, or state must be protected (locks / concurrent collections / immutable data)
- Must **NOT** depend on **scoped** services (like DbContext) directly

Common mistake:

Registering a singleton that uses EF Core DbContext.

DbContext is **Scoped** (per request). If a singleton holds it, you'll get runtime errors or subtle bugs.

Real-world singleton examples in ASP.NET

1) Caching provider (application-wide cache)

- Example: IMemoryCache (built-in) is typically used as an app-wide singleton service.
- You cache product lists, configuration, feature flags, etc.

Why singleton fits: cache should be shared across requests.

---

2) HttpClient / typed clients (shared connection pool)

In ASP.NET Core you normally use IHttpClientFactory, but the *goal* is the same: avoid creating a new HttpClient per request.

Why singleton-ish fits: connection management is expensive; reuse improves performance and avoids socket exhaustion.

---

3) Configuration / feature flag service

A service that reads app settings and provides strongly-typed access.

Why singleton fits: settings don't change often; reading once is efficient.

---

4) Thread-safe in-memory sequence generator (rare, but real)

Example: generating unique IDs for temporary in-memory work (not database IDs), or rate-limiting counters.

Why singleton fits: it must coordinate across all requests.

(But be careful: in multi-instance deployments, each server has its own singleton.)

5) Shared “policy” objects (rare)

Stuff like:

- Validation rule registries
- Mapper configuration (AutoMapper config is effectively singleton-ish; mappings are built once)

When Singleton is a bad idea in ASP.NET

Anything request-specific

- Current user, claims, request headers, culture, etc.
  Those are **scoped** (or transient) concerns.

Anything that holds mutable state per user

Singleton state is global. If you store user data there, users will leak into each other.

Anything that is not thread-safe

If multiple requests mutate it concurrently, you can corrupt state.

Anything needing DbContext

Use scoped services or factories.

“Singleton” in load-balanced / cloud apps

Even if you register as singleton, it’s:

- **one per process**
- not one per “system”

If you run 5 instances in Kubernetes/Azure/App Service scale-out, you have **5 different singletons**.

So, singletons are great for **in-process** stuff (cache, configuration), not for “global truth” (that’s what databases/redis are for).