IT Project Guidance

On Caching

Version:

0.1

## Purpose

This guidance introduces caching as a foundational design decision for modern web services. It provides a pathway to begin with lightweight in-memory caching and evolve toward distributed caching architectures as scale demands. The guidance targets civic platforms and early-stage services that seek growth without premature infrastructure expense.

## Synopsis

Caching is no longer a performance optimisation—it is a baseline expectation. This document outlines strategies for implementing caching from the outset, avoiding rework as systems mature. It provides practical examples, architectural patterns, and design principles for designing scalable and cost-effective caching layers. Emphasis is placed on beginning with what is needed now, while enabling seamless growth into more complex patterns as required.

## Contents

[Purpose 1](#_Toc201924823)

[Synopsis 1](#_Toc201924824)

[Contents 2](#_Toc201924825)

[Purpose and Audience 3](#_Toc201924826)

[Scope 3](#_Toc201924827)

[Background 3](#_Toc201924828)

[Objective 3](#_Toc201924829)

[What’s Being Cached Today in Big Systems 3](#_Toc201924830)

[Types of Caching 4](#_Toc201924831)

[Starting Correctly 4](#_Toc201924832)

[Starting Small 4](#_Toc201924833)

[Ready to Grow 5](#_Toc201924834)

[Caching Principles 5](#_Toc201924835)

[What to Cache on CDN 5](#_Toc201924836)

[What to Cache via Service 5](#_Toc201924837)

[What not to Cache 5](#_Toc201924838)

[Exit Strategy 6](#_Toc201924839)

[Conclusion 6](#_Toc201924840)

[Appendices 7](#_Toc201924841)

[Appendix A - Document Information 7](#_Toc201924842)

[Versions 7](#_Toc201924843)

[Images 7](#_Toc201924844)

[Tables 7](#_Toc201924845)

[References 7](#_Toc201924846)

[Review Distribution 7](#_Toc201924847)

[Audience 7](#_Toc201924848)

[Structure 7](#_Toc201924849)

[Diagrams 8](#_Toc201924850)

[Acronyms 8](#_Toc201924851)

[Terms 8](#_Toc201924852)

# Purpose and Audience

This document is for developers, architects, and service owners designing scalable systems from the outset. It emphasises intentional caching strategies for services that aim to support dynamic workloads, evolving privacy models, and operational cost constraints.

# Scope

The guidance covers core caching strategies suitable for APIs, data services, and frontend platforms. It includes in-memory caching, distributed caches such as Redis, and transitions to policy engines. It provides implementation considerations specific to .NET but is conceptually applicable across environments.

# Background

Caching, like security, is not a bolt-on capability. It is a foundational architectural concern that must be present from the beginning if a service intends to be reliable, responsive, and capable of scaling. Services that treat caching as an afterthought often find themselves re-architecting at the worst possible time—when demand is already increasing and responsiveness is falling short.

Delaying caching strategy design and implementation may appear cost and time saving in early development stages, but such short-term frugality can result in catastrophic long-term loss. Many civic and commercial platforms initially took a local, low-scale approach to system responsiveness, only to find themselves unable to serve regional or international audiences quickly enough. These delays allowed competitors to detect market demand, enter aggressively, and surpass the original innovator.

Caching enables responsiveness. Responsiveness in turn enables availability. The quicker a system responds, the more users it can support simultaneously, and the broader the audience it can serve. While caching is not the only factor in scalable delivery—sharding, distribution, and synchronisation are also vital—it is the one that unlocks early growth without rewriting core logic.

Despite its importance, caching is frequently misunderstood or underused. Developers may get limited exposure to caching patterns early in their careers, especially when in-memory management and performance tuning are abstracted away. Access to cache infrastructure requires deliberate architecture and memory awareness, both of which are often deferred or delegated.

This document aims to close that gap by explaining the types of data typically cached, the principles of cache management, and patterns for beginning with in-memory approaches and evolving toward distributed systems. Examples of caching strategies in large-scale systems have been moved to the appendices, where they provide comparative context but do not distract from the core guidance.

# Objective

The objective is to design systems that cache from day one without locking into cost or complexity. Services should remain flexible, able to operate cheaply in early stages but grow into enterprise patterns without rewriting core logic.

# Purpose

Caching exists to reduce latency, eliminate redundant processing, and ensure responsive service delivery by retaining ready-to-use, formatted data in fast-access memory. Rather than repeatedly querying persistent storage and reprocessing unchanged logic or data structures, caching allows systems to serve frequently accessed information directly from memory. This shift not only decreases response time but also shields upstream systems from unnecessary load, improving both user experience and infrastructure longevity.

The concept of 'rarely changes' is context-dependent. Different types of information evolve at different rates: configuration values may stay constant for months, system settings almost as constant, user settings might change only occasionally, while session preferences or roles could update more frequently. However, even the most dynamic of these typically change at a pace vastly slower than individual user requests. Caching takes advantage of this asymmetry, allowing data to be served efficiently until it becomes outdated enough to warrant recomputation or refresh.

# What to Cache

Not all of the following has to be cached, but

# What’s Being Cached Today in Big Systems

Examples include:

* Instagram: user feeds, story views, permissioned media visibility
* TikTok: per-user watch history, feedback scoring, region-based visibility rules
* Microsoft: tenant-level feature toggles, authentication token metadata
* Education platforms: visibility rules, enrolment status, filtered content views These systems cache at the attribute level—per user, per object, per session. Tokens, visibility rules, computed views, and consent evaluations are common candidates.

Sizes include:

# Types of Caching

In Memory

Redis

CDN

# Starting Correctly

As with all OO based approaches, start by developing a Service (e.g. CachingService) that provides needed methods to wrap a framework or library, isolating it from the rest of the app, so that when you upgrade, change implementation of cache, you don’t have to rewrite it everywhere.

The service wraps a dictionary of key-to-function pairs, configured at startup.

The primary method accepts just a key, to find the right function, then invokes it.

The function tests whether its value is expired, returning the current value if valid or refreshing it if not.

This protects the code base from having lots of different places where the cache is tested, and reset – potentially with different time durations, etc.

This defers recomputation until needed, without bloating startup time.

Functions should be deterministic and side-effect free. Restarting the app will reset values, so consider seeding from known state or default values. Use caution with async factories and long timeouts.

# Starting Small

Building a Cache-Aware Service

Begin with in-memory caching.

In .NET, IMemoryCache is available out-of-the-box.

Structure your keying carefully: include user ID, audience type, object ID, and attribute. Implement TTL and sliding expiration for control. Invalidate entries on relevant upstream changes. Focus initially on per-request optimisations: cached consent decisions, user role lists, and data filtering rules.

Intermediate Pattern: Cache Function Dictionary

# Ready to Grow

When your service becomes scalable, all you’ll have to do is replace the memory based implementation with a Redis based on.

No other code requires changing.

# Caching Principles

Cache as close to use as possible

Cache computed views, not raw data

Cache in the user’s natural language/context

Use short TTLs for frequently updated items (e.g. 5s for feed filters)

Avoid precomputing if request rate is low

Secure values (e.g. role claims, consent results) may be cached server-side but never round-tripped to the client

Use persistence only where refresh is costly or change is infrequent.?

Avoid orchestration patterns unless request volume justifies.

Cache-aside patterns remain most efficient and predictable?

# What to Cache on CDN

unchanging artefacts

# What to Cache via Service

Consent rule evaluations

Token introspection or validation results

User-specific view DTOs

# What not to Cache

Raw database rows

Data tightly coupled to business logic changes

Anything frequently invalidated by background processes

# Exit Strategy

No idea what this section is about?

Turning Caches into Policy Long-term, some cache structures evolve into business logic:

* Consent results → OPA policies
* Visibility rules → Feature flags
* DTO projections → View templates Once mature, these can migrate to formalised policy engines. By caching with semantic clarity early, this transition becomes simple.

# Conclusion

Caching is not a performance enhancement—it is foundational. Even small services benefit from well-structured caches. By starting simple and planning ahead, systems grow without rework. A good cache is not a shortcut. It is a reflection of clear thinking and well-bounded logic, delivered fast.

Appendices

Appendix A - Document Information

Authors & Collaborators

* Sky Sigal, Solution Architect

### Versions

* 1. Initial Draft

### Images

[Figure 1: TODO Image 2](#_Toc144995112)

### Tables

[Table 1: TODO Table 3](#_Toc145048484)

[Table 2: TODO Table 2 3](#_Toc145048485)

### References

**There are no sources in the current document.**

### Review Distribution

The document was distributed for review as below:

|  |  |
| --- | --- |
| Identity | Notes |
|  |  |
|  |  |
|  |  |

### Audience

The document is technical in nature, but parts are expected to be read and/or validated by a non-technical audience.

### Structure

Where possible, the document structure is guided by either ISO-\* standards or best practice.

### Diagrams

Diagrams are developed for a wide audience. Unless specifically for a technical audience, where the use of industry standard diagram types (ArchiMate, UML, C4), is appropriate, diagrams are developed as simple “box & line” monochrome diagrams.

### Acronyms

API

: [Application Programming Interface](#Term_ApplicationProgrammingInterface).

DDD

: Domain Driven Design

GUI

: [Graphical User Interface](#Term_ApplicationProgrammingInterface). A form of [UI](#Acronym_UI).

ICT

: acronym for Information & Communication Technology, the domain of defining Information elements and using technology to automate their communication between entities. [IT](#Acronym_IT) is a subset of ICT.

IT

: acronym for Information, using Technology to automate and facilitate its management.

UI

: User Interface. Contrast with [API](#Acronym_API).

### Terms

Refer to the project’s Glossary.

Application Programming Interface

: an Interface provided for other systems to invoke (as opposed to User Interfaces).

Capability

: a capability is what an organisation or system must be able to achieve to meet its goals. Each capability belongs to a domain and is realised through one or more functions that, together, deliver the intended outcome within that area of concern.

Domain

: a domain is a defined area of knowledge, responsibility, or activity within an organisation or system. It groups related capabilities, entities, and functions that collectively serve a common purpose. Each capability belongs to a domain, and each function operates within one.

Entity

: an entity is a core object of interest within a domain, usually representing a person, place, thing, or event that holds information and can change over time, such as a Student, School, or Enrolment.

Function

: a function is a specific task or operation performed by a system, process, or person. Functions work together to enable a capability to be carried out. Each function operates within a domain and supports the delivery of one or more capabilities.

Person

: a physical person, who has one or more Personas. Not necessarily a system User.

Persona

: a facet that a Person presents to a Group of some kind.

Quality

: a quality is a measurable or observable attribute of a system or outcome that indicates how well it meets expectations. Examples include reliability, usability, and performance. Refer to the ISO-25000 SQuaRE series of standards.

User

: a human user of a system via its UIs.

User Interface

: a system interface intended for use by system users. Most computer system UIs are Graphics User Interfaces ([GUI](#Acronym_GUI)) or Text/Console User Interfaces (TUI).