🏷️ Cache Connection & Authentication Design Documentation

🎯 Goal

Design a flexible and extensible system to support multiple cache types (Redis, Memcached, Hazelcast, Ignite, etc.) and multiple authentication mechanisms (username/password, token, TLS, IAM, etc.), while minimizing changes to core classes as new fields or cache/auth types are added.

1. Core Design Principles

- **Open/Closed Principle (OCP):** Classes should be open for extension but closed for modification.
- **Single Responsibility Principle (SRP):** Each class handles one concern (cache type config, auth config, etc.).
- **Factory Pattern:** Dynamically create the correct config/connection objects based on cache type or auth type.
- **Strategy Pattern:** Plug in different authentication mechanisms easily.
- **Composition over Inheritance:** Authentication config is composed inside cache connection config.
- **Polymorphism:** Handle different cache/auth subclasses via their base class.

2. Cache Connection Configuration

Base Class

- **Class:** BaseCacheConnectionConfig
- **Purpose:** Defines common fields for all cache types.
- **Fields (common across all caches):**
  - cacheType (enum: REDIS, MEMCACHED, HAZELCAST, IGNITE, etc.)
  - host
  - port
  - useSSL
  - isAuthRequired
  - authConfig (polymorphic, see Authentication Configuration)

Cache-Specific Subclasses

Each cache type extends the base class and adds its own extra fields.

- **RedisConnectionConfig:** databaseIndex, isCluster, clusterNodes, sentinelNodes, masterName
- **MemcachedConnectionConfig:** hashAlgorithm, maxConnections
- **HazelcastConnectionConfig:** clusterName, discoveryMode, memberList
- **IgniteConnectionConfig:** discoverySpi, communicationSpi, persistenceEnabled

Pattern Used: Inheritance for specialization.

Factory Used: CacheConfigFactory creates the correct subclass based on **cacheType**.

3. Authentication Configuration

Base Auth Class

- **Class:** AuthConfig
- **Field:** authType (enum: NONE, USERNAME_PASSWORD, TOKEN, MTLS, KERBEROS, CLOUD_IAM)
- **Purpose:** Defines a contract for all authentication strategies.

Auth Subclasses (Strategy Pattern)

- **UsernamePasswordAuthConfig:** username, password
- **TokenAuthConfig:** token
- **MtlsAuthConfig:** certPath, keyPath, trustStorePath
- **KerberosAuthConfig:** principal, keytabPath, realm
- **CloudIamAuthConfig:** provider (AWS, GCP, Azure), roleArn or serviceAccountId

Factory Used: AuthConfigFactory creates the right auth object based on **authType**.

4. Integration of Cache & Auth

- Every cache connection config contains:
    - **Common base fields**
    - **Polymorphic authentication config**
    - **Cache-specific fields**
- Flow:
    1. Service provides a JSON/YAML config.
    2. CacheConfigFactory reads cacheType → creates subclass.
    3. AuthConfigFactory reads authType → creates auth strategy.
    4. Connection factory establishes the cache connection using both.

5. Field Evolution Strategy

Purpose

Define how to handle new fields in cache connection configs so the design remains extensible, maintainable, and backward-compatible.

Three-Tier Strategy

A. Typed Fields (Stable & Critical)

- Fields that are core and widely used.
- Example: host, port, sslEnabled, databaseIndex (Redis).
- Rule: Add only if field is stable and critical to the cache type.

B. Extra Properties (Flexible Extension Point)

- Generic key-value store inside the class (extraProperties) for experimental or vendor-specific options.
- Example: readFromReplica, compressionLevel, connectionTimeout.

- Rule: Place new/unproven fields here first. Promote to typed fields when stable.

C. Subclassing (Radical Divergence)

- Create a subclass only if a cache type needs **significantly different behavior/config**.
- Example: RedisClusterConfig subclass for cluster-only properties.

Patterns Used

- **Composition:** extraProperties allows flexible extensions without modifying base class.
- **Open/Closed Principle:** Base classes remain closed for modification; extensions use extraProperties or subclassing.
- **Builder Pattern:** Recommended for creating objects with optional extra fields.
- **Promotion Path:** Extra fields → typed fields → subclass (if divergence occurs).

6. Benefits

- **Flexibility:** Quickly support new cache features via extraProperties.
- **Safety:** Core fields remain strongly typed and validated.
- **Maintainability:** No need to refactor core classes for every vendor update.
- **Future-Proofing:** Clear lifecycle for field evolution ensures long-term extensibility.

7. Field Evolution Lifecycle Diagram

Flow Summary:

1. New field appears → goes into extraProperties.
2. If usage becomes stable → promoted to typed field.
3. If field causes radical divergence → subclass created.