Persistence

The persistence layer is responsible for insulating various components that have to preserve data across multiple web requests from the specifics of data storage. This includes data associated with specific clients as well as data that has to be globally accessible when servicing all clients.

Overview

The core storage interface used by the code base is org.opensaml.storage.StorageService, and the package includes a few other public interfaces that complement it. At the present time, only a single use case for storing data is implemented with a custom API (the JDBC implementation of SAML PersistentID generation). Everything else relies on this interface.

Storage tends to be a controversial topic with developers, all of whom believe in their deepest souls that they know best and that their storage solution is the right one. The API we provide is not fancy, or particularly elegant, and it can make life hard when developing new components that need to store data. But close to a dozen storage use cases have all been implemented with it, with near total portability to any of the implementations provided with Shibboleth or provided by third parties.

In a few cases, components rely on capability detection to alter their behavior when faced with limited storage features, but for the most part deployers are expected to configure the right "kind" of storage plugin for various use cases.

Features

The following summarizes the capabilities of the API:

- String-Based API
  - Handles storing string and text data (blobs must be encoded as text), keeping serialization of objects as a separate consideration from storing them. One of the consequences of this is that object aliasing and graphs have to be implemented by hand by managing alternate indexes to information. For example, a secondary key B to an obect keyed by A would be stored as a mapping of the strings (B, A) so that B can be used to find A. If the mapping of B to A is not unique, then the value becomes a list requiring upkeep, and this can cause performance problems if the set of A is unbounded or large, so this has to be managed carefully.
- Two-Part Keys
  - Records are stored under a two-part key that emulates the concept of a storage context or partition of records. This makes it practical to share one instance of a storage back-end across different client components. Some back-ends do not support this explicitly and are forced to emulate it.
- Exposing Capabilities
  - Exposing back-end implementation capabilities such as maximum key size enables clients to intelligently query for them and adapt behavior. For example, some components might be able to truncate or hash keys while others might not.
- Internal Synchronization
  - All operations are 100% ACID: threading is not a consideration and implementations are expected to perform internal locking and transactional boundaries as required to guarantee this to callers. Performance is **not** the priority, safety is.
- Versioning
  - All records are versioned. Attaching a simple incrementing version to records makes detecting collisions and resolving contention relatively simple without necessarily losing data. Callers can determine whether to ignore or reconcile conflicts based on their needs.
- TTLs
  - All records can have a TTL value to support efficient cleanup. Some components make creative use of the TTL value to embed information about the record that can be recovered by calculating an offset from the TTL.