Profile Example Microservice

v1.0 ©2019 Sean Donn, MIT license

Table of Contents

Requirements

In a microservice architecture for e.g. a financial institution, there is a need for a service to store customer profile data in a secure fashion, with appropriate access controls and audit logging.

1. Data can be stored, updated, and retrieved at any point of time under a universal ID
2. Should be accessed via HTTP
3. Data update authorization required
4. Data is safely secured and accessible based on permission details
5. Service clients include other microservices
6. Should have knowledge about who updated the information

Design Considerations

Features

1. Data is not required to follow a specific schema and will be stored as schema-less JSON objects
2. Permissions control will be enforced at the first level attribute in the JSON object to allow for multiple client services with varying permissions
3. Use JWT for authentication for service calls and use claims to hold permissions.
4. Use version 1 UUID for future scalability via partitioning into multiple nodes
5. Embedded key-value store for demonstration purposes
6. Use JSON Patch standard for updates
7. Use JSON Path specification for querying (vs. GraphQL which requires a schema)

Note: For demonstration purposes a secret key is used for JWT, production usage should leverage certificates and store them in something more secure like Hashicorp Vault. Also the data encryption key is stored in the properties file which should also be stored similarly. Further, the service is HTTP-only for demonstration, and should be deployed as HTTPS with proper certificates in production.

Assumptions

1. JWT tokens are issued by a separate microservice in charge of authentication and authorization
2. Non-functional requirements on data size, service level availability, performance, etc. are not taken into account, however, see section on scalability.
3. Security configuration is minimal for demonstration purposes, see prior section notes for production deployment considerations.
4. Distributed transactions are typically not available in microservices architecture so support for optimistic locking is required.
5. Updates to objects are infrequent enough and disk space continues to be cheap enough to allow storing of all changes.

Design

Access Control

Permission claims embedded in the JWT controls access to data and the API. The claim type is `Perm` and the value is of the form <level>:<read/write/history flags>. There are three levels of permissions:

1. Global service: the `profile` permission is required for read or write access
2. Profile object: the `profile.<uuid>` permission controls access to an object with the given uuid, or * for all uuids
3. Profile attribute: the `profile.<uuid>.<attribute>` permission controls access to the specified top-level attribute in the profile JSON object

The three flags are

1. `r` – read
2. `w` – write
3. `h` – history read

Examples:

`profile.*.*:rw` is a global wildcard permission to read and write any attribute in any profile object
`profile.d6e8694c-0fea-11e9-b0ec-01c0f03d5b7c.foo:r` allows reading of the attribute `foo` in the profile object `{ "foo": "123", "bar": "234" }` stored under UUID `d6e8694c-0fea-11e9-b0ec-01c0f03d5b7c` but does not allow reading of `bar`
`profile.*:h` allows access to history data for any profile object
`profile:w` will be required to create new profiles

API Table of Operations and Required Access

Operation	Permissions Required	REST Request	Results
List all profile IDs*	`profile:r`	GET /service/profile	401 – unauthorized 200 – list of profile IDs
Create new profile	`profile:w` `profile.*.<attribute>:w`	POST /service/profile body: JSON data	401 – unauthorized 200 – new UUID of the stored profile
Fetch profile	`profile.<id>:r` `profile.<id>.<attribute>:r`	GET /service/profile/<id>	401 – unauthorized 200 – JSON of only permitted top-level attributes

Replace profile*	`profile.<id>:w`
	`profile.<id>.<attribute>:w`
	`profile.<id>.<attribute>:r`
	if using predicate guard
		PUT /service/profile/<id>
		X-predicate: JSONPath
		body: JSON data
			401 – unauthorized
			
			304 – not modified due to predicate failing
			
			200 – update successful

Update profile	`profile.<id>:w`
	`profile.<id>.<attribute>:w`
	`profile.<id>.<attribute>:r`
	for <attribute> that are part of a move, copy, or test operation
		PATCH /service/profile/<id>
		body: JSON patch
			400 – bad request due to invalid patch format
			
			401 – unauthorized
			
			304 – not modified due to test op failing
			
			200 – update successful

Query profile*	`profile.<id>:r` `profile.<id>.<attribute>:r`	POST /service/profile/<id>/query body: hash of JSONPath queries	401 – unauthorized 200 – hash of JSON Path results
Delete profile*	`profile:w`	DELETE /service/profile/<id>	401 – unauthorized 200 – success
Query profile history*	`profile.<id>:h`	GET /service/profile/<id>/history	401 – unauthorized 200 – list of history objects

* not implemented – exercise for the reader. For list profile IDs and query profile history, query parameters to specify window sizes for the results would be recommended.

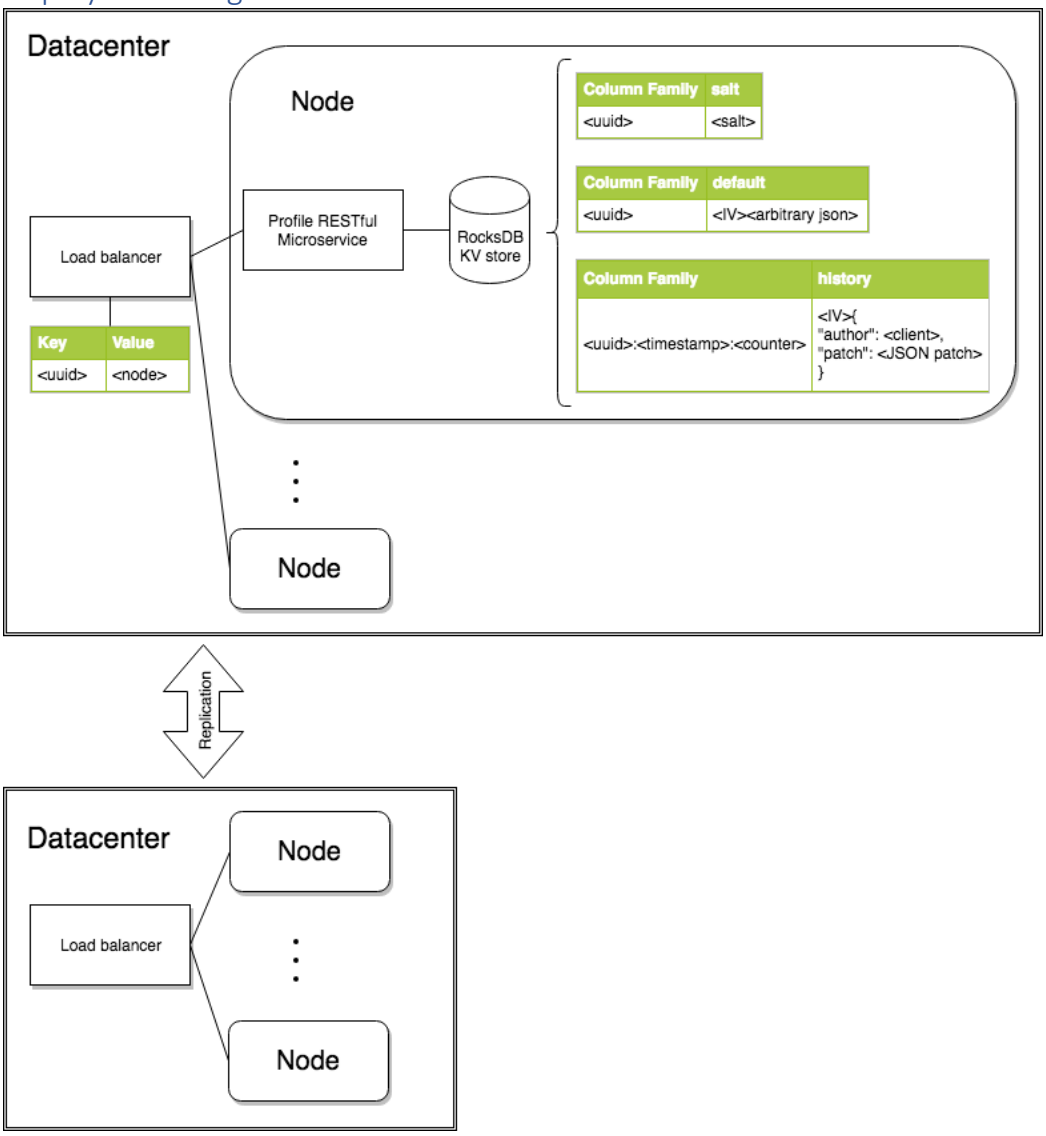
Optimistic Locking

The update/replace (write) operations support optimistic locking via two forms. The replace operation replaces the entire body of the profile object and supports a custom header specifying a JSON Path expression which can use predicates (see e.g. https://github.com/json-path/JsonPath#predicates). If the JSON Path expression evaluates to a non-empty array or non-empty value then the replace operation will proceed.

For the update operation, JSON Patch already specifies the test operation as a mechanism which will fail the update if the test operation is not successful.

Typical use of optimistic locking will use some sort of version number that is retrieved on read and then that version number will be specified when writing, so if the version number is altered the update will fail and thus avoid overwriting another client's update. On successful write the version number is incremented.

Deployment Diagram



Data Storage

At the node level, data is stored in RocksDB, an efficient key-value store available as a library. RocksDB has the concept of column families, which allows for storage of different types of data with the same or similar key.

UUID Generation

Each profile object is keyed by a type-1 UUID https://en.wikipedia.org/wiki/Universally_unique_identifier#Version_1_(date-time_and_MAC_address), which includes the node ID to avoid the need for nodes to coordinate at run time to guarantee uniqueness. The UUIDs are simply opaque from a client perspective and no authentication or authorization is implied by the UUID.

Encryption

While transport level (data in transit) HTTPS encryption and disk-level (data at rest) encryption should be used, modern systems run 24x7 and are at risk of compromise while the system is running, so application-level encryption provides another level of security. To avoid catastrophic compromise of all data and provide resilience against known plaintext attacks, the AES encryption key is salted via PBKDF so that each profile object is encrypted with a different key. In addition, GCM mode is used for the AES cipher and a different initialization vector (IV) is stored with each encrypted value. The 8-byte salt is stored in a separate column family from the profile object itself, and the 12-byte IV is stored at the beginning of each encrypted value: the profile object and all of its history entries.

History

As there can be multiple history entries per profile, the entries have keys of the form `<uuid>:<Unix millisecond timestamp>:<counter>`, which maintains lexical ordering and can also be queried via time window. The counter is necessary in case the timestamp does not have enough practical resolution to disambiguate entries.

Each history entry records the author of update, which is taken from the Subject of the JWT token authorizing the update. The content of the entry is a JSON patch against the prior version of the profile object, or the empty object `{}` if the entry is the first one. This design provides an audit record of every change and can be used to replicate changes in a highly-available environment. If disk space or performance becomes an issue, the history can be trimmed in-band or via an out-of-band process.

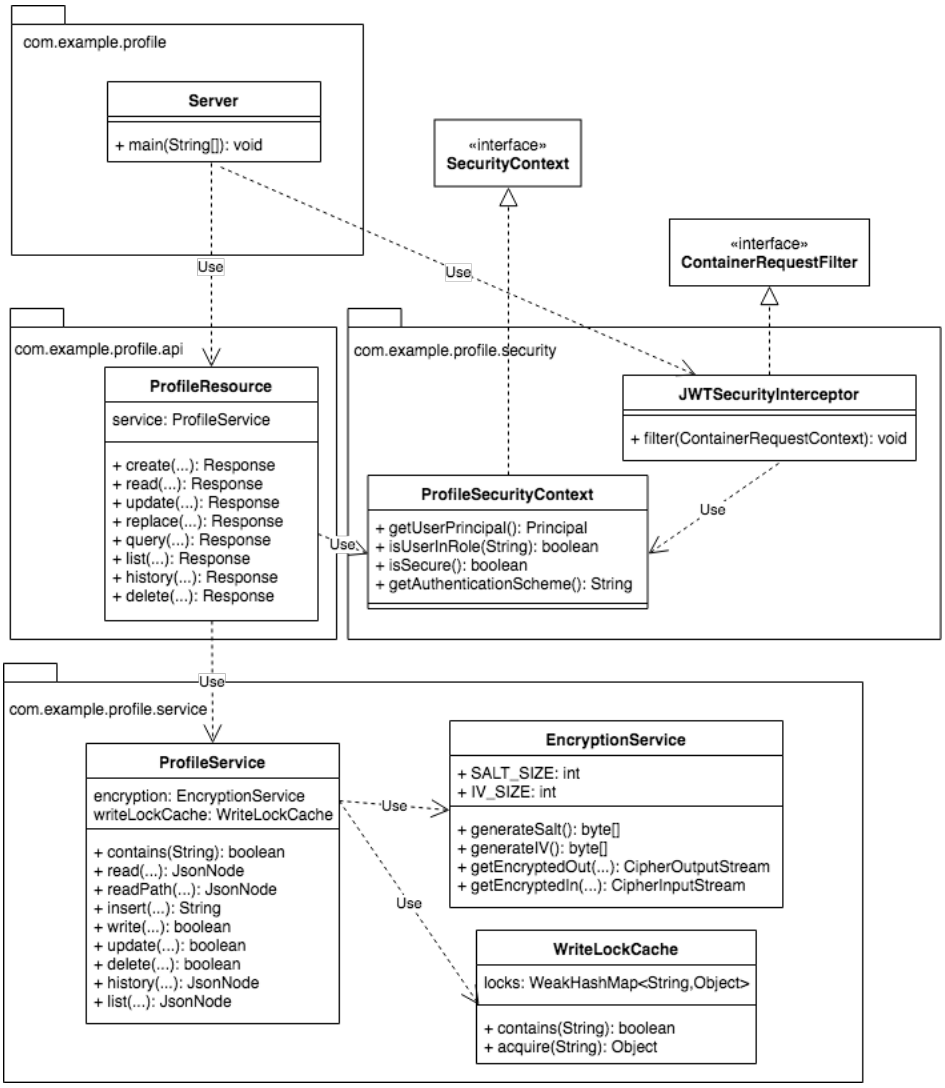
Scaling

The system is trivially scalable horizontally by adding a load balancer in front of multiple nodes. A simple balancing scheme can examine the UUID for the node ID and route to the appropriate node for operations with a UUID, or round-robin otherwise. With a larger dataset this will eventually cause load imbalance among the nodes, so a key-value store mapping UUIDs to nodes would better balance the load. Further enhancements could update this store via real-time feedback of the load amongst the nodes.

To scale beyond a single datacenter, an HA failover pair of the form of a hot and warm standby can be implemented by replicating the history entries over a message queueing system to the standby datacenter. If consistency guarantees are relaxed to allow eventual consistency, this method can also allow for multiple hot datacenters, where each node ID will have a primary datacenter “home” that publishes updates to the other datacenters, and reads from the datacenters are guaranteed to be eventually consistent.

For stronger consistency guarantees, replication of updates in-band with the API call will be necessary. At this point however it is more advisable to switch the data storage to a distributed key-value store with the desired consistency properties, rather than extending this design.

Class Diagram



Code Overview

The JAX-RS RestEasy framework was picked to provide the API interface, along with Weld CDI for the injection framework, due to familiarity and time-constraints. Security is implemented via a JAX-RS request filter that pulls the JWT token from the Authorization header and sets a custom security context that is able to evaluate access control against the JWT permissions claims as described previously.

For more detailed specifications, please read the spec-style test cases.

Class	Description
Server	Main class that boots up RestEasy with JWTSecurityInterceptor and ProfileResource

JWTSecurityInterceptor	Decodes the JWT authorization token and sets the ProfileSecurityContext for each request
ProfileSecurityContext	Custom SecurityContext which overrides isUserInRole to evaluate access control against claims in the JWT
ProfileResource	The JAX-RS resource that implements the Restful API, enforces access control by checking with the ProfileSecurityContext, and invokes the appropriate methods in ProfileService
ProfileService	Service class that encrypts data using EncryptionService, enforces write locks, and calls RocksDB library for data storage and retrieval
EncryptionService	Provides salt and IV generation, and creates appropriate input/output cipher streams
WriteLockCache	Simple cache of objects used as write locks by ProfileService to serialize updates to the same profile object