`https://github.com/JonathanEarle/CanonicalAuditLogger`

Canonical Technical Assessment
Design & Architecture Documentation

Jonathan Earle

This document gives a brief overview of the architectural and design decisions made for an audit logging service which accepts event data sent by other systems and provides an HTTP endpoint for querying recorded event data by field values. The service is based on an entity-event model where each type of entity has a set of event types it can perform and events have a set of defined attributes and a set of variate attributes set by the user. The database is utilized to enforce adherance to this model. Docker was used for container management of the database and audit server. The audit logger was developed as a proof of concept and thus HTTPS support was not built into the server. In addition to a lack of HTTPS support the server was implemented using python's http server modules which is not made for production and a suitable alternative would need to be sourced for production viability. Finally this document discusses a few of the design decisions made during the implementation of the audit logging service.

Contents

1. Architectural Decisions

A RESTful architecture was chosen to implement this service due to its flexibility in terms of the data it can receive and send as well as the resources and methods it can use.

In this service every event is performed on a certain entity this model was chosen to simulate the real world. It defines a stricter model for the user to adhere to to ensure the data remains logically consistent. Having such a strict model means more work is done initially by the user to define the model. An alternative method would be to have no strict adherence to definitions such as this, but a system with too little constraints is susceptible to denormalization.

The creation and modification of entity and event types is a meta process as the execution of these tasks themselves utilize the main event auditing module.

Every time an action is performed on a new entity it is logged and stored in an entity instance table which is updated each time a change is made to that particular instance. This can be used to store the current state of each entity in the system and thus can function as a lookup of the system's state.

The architectural model of the authorization was basic username password authorization coupled with the creation of a bearer token for access. This model should be re-engineered in the future to implement OAUTH 2.0 as this is the industry standard.

Finally the service is deployed within two docker containers which host the postgres database and the python service.

2. Database Architectural and Design Decisions

The system is ultimately an event sourcing architecture in action, which relies on recording all changes made to an application. Utilizing a relational database was an advantageous design choice because they have inherent referential integrity rules and rigorously enforce them, which can be easily utilized to enforce an entity event model. Relational databases also are tabular in nature which is ideal for storing a list of atomic events.

Postgres was chosen due to its wide array of specialized tools which were leveraged to create slightly more complex queries and can be utilized in the future in the event the service's requirements become more complicated. Utilizing the database's built in constraint management to enforce the integrity of a model takes some processing load away from the application. Postgres is also is resilient to performance issues when performing write intensive operations

One prime example which show cases the decision to use the database to enforce the entity event model is in the edit_entity_events method of the EventType class. The ability for an entity to perform an event is stored as a two column table of the entity type id and event type id. Referential integrity rules are taken advantage of to determine if an entity can perform a particular action.

The database was however only selectively used in this manner for example in the editing of an event type's attributes a python set is used to ensure no duplicates and to modify the existing list of attributes. This contrasts the previous implementation despite the similar nature of their actions.

3. Design Decisions

Many design decisions went into developing this service; while they cannot all be documented at a granular level as it would mainly be trivial, listed below is the summary of a few noteworthy designs implemented.

3.1. Validation on request:

Validation of received data is done only at the point where it is needed the routers, server or authorization process does not validate the data This compartmentalization of data validation means no single validator has to handle many different types of requests or that many different validators need to be tracked as they are all held within the required method.

3.2. Stored procedures & Functions

Postgres stored procedures and functions were used to speed up execution of repetitive actions such as adding a new event and checking whether an entity can perform specific events.

3.3. Uniformed messages

Consistent use of the response format response message, success and http code which compactly and uniformly delivers all responses from the server. This generalization within the system results in easy managing of server response.

3.4. Routing table

A routing table of key value pairs was used to easily look up the method each endpoint must perform, this design is also human readable and easy to extend.

3.5. Generalized errors

The data returned to the user is generalize to prevent unnecessary data leakage by external exceptions being raised. Such as database error revealing information on the database contents.

3.6. Version targeting

The version of each endpoint is targetted to anticipate incompatible system changes of the same funtion in future versions.

3.7. User account initialization

User has events and entity relationship model defined on user initialization this gives them the ability to use the event type and entity type modelling methods.

3.8. Ignore unnecessary data

Extra parameters sent in by the user are ignore as it would be a waste of processing time to acknowledge them.

3.9. JSON encoding

JSON data was used as the encoding method for requests and responses as it is easy to read, work with and is also widely used.

3.10. Single authorization instance for each request

Authorization object persists through the life cycle of the request, this is done to verify at multiple points during the routing process that this action can be performed.

3.11. Read only properties

Particular properties were presented as read only such as the status of authorization, authorized user and the type of authorization too notify other developers these fields should not be tampered with.

4. Server Design Decisions

HTTPS was not used as this is a local testing environment but would be utilized in a real world case. Reasons for not utilizing HTTPS include:

1. No secure cookies were set
2. No mixed content
3. No 3rd party APIs requires it

The simple http python server is not recommended for production as it only implements basic security checks an alternative such as Gunicorn should be used behind a proxy instead.