Audit Service - Design Document

Introduction

This document will cover the flow to design an audit service for a microservices-based application, it involves several components and considerations. This document will cover the break down the problem statement a low-level design for requirement:

Requirements

Functional Requirements

Below is the requirement to create an audit service for microservices-based applications.

In Scope

Below is the in scope function requirement

- Multiple service publish message
- The audit service should subscribe to change notifications from all other publisher services, process and save them into a standard format in database
- View audit message APIs:
    - Role based access control
    - Admin can view all the audit events
    - Non admin can access only
- Log retention policy

Considerations:

1. Database of choice and schema design with considerations around performance
2. Format of the audit message
3. Intra-service communication
4. Safeguard against audit tampering
5. Considerations for a cross-platform deployment
6. Considerations for a scalable deployment

This design document is made assuming the below points -

1. Audit service will expose REST based apis
2. Database: There is no search use case for audit logs, so no requirement for non-structured data
3. Asynchronous process to subscribe notification to audit service
4. Audit service will contain only modified data, not complete logs details

Design would require modifications If any of the above assumptions is changed.

Out of Scope

Below are actions are not considered

- Searching on
    - Only on Audit event database columns
    - Not on event details
- View audit message apis:
    - JWT token expiry
    - Pagination
    - Filtering on service name

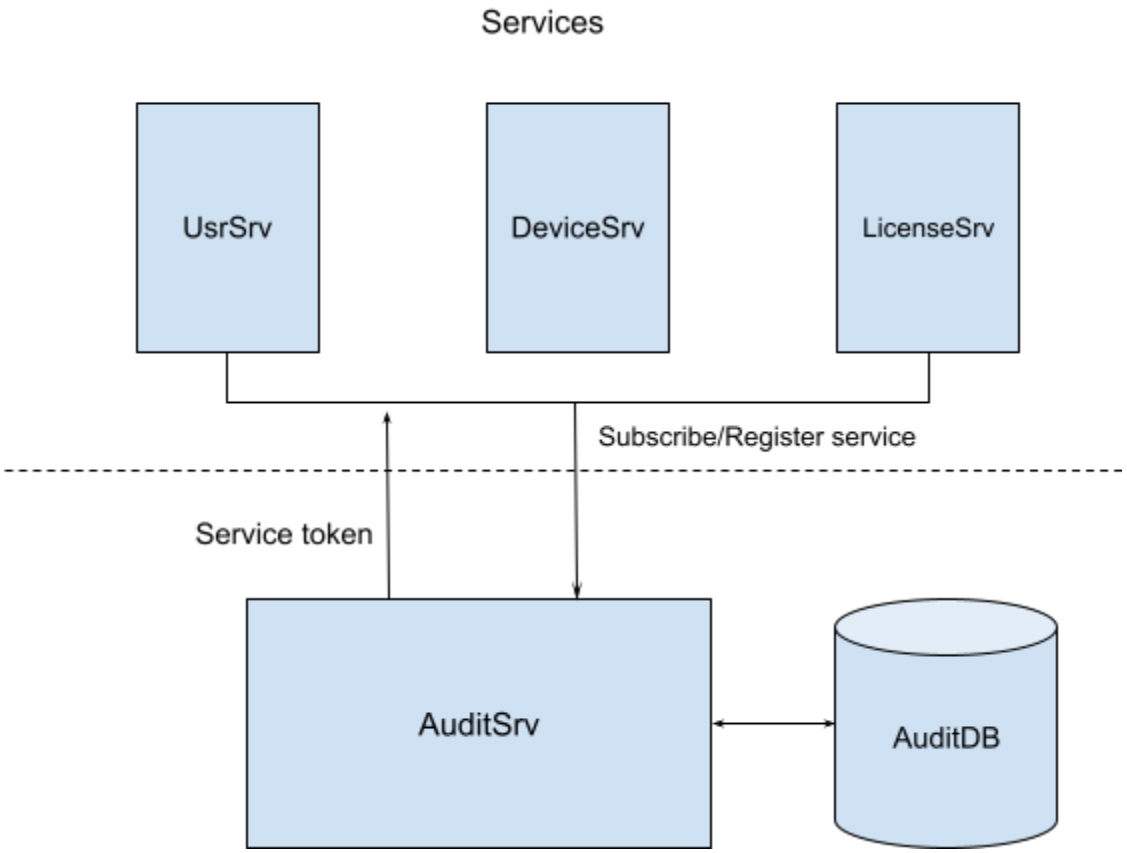
Workflows

Subscribing to Change Notifications

There are two approaches to achieve this

1) Synchronous
2) Asynchronous

Approach #1 : Synchronous
Every service must subscribe/register to audit service before pushing changes to audit service, audit service will provide a token to individual services that will use it to accept change from services. Below is the high level design workflow of this approach



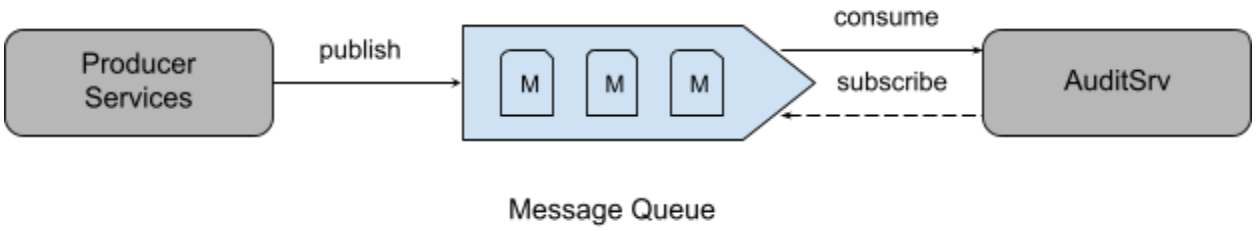
Approach #2 : Asynchronous (Implemented)

We can use a message broker or event streaming platform, such as Amazon SNS/SQS, Apache Kafka or RabbitMQ, that enables asynchronous communication between microservices. Whenever a microservice makes changes to an entity, it will publish an event containing the necessary information about the change. The audit service will then subscribe to these events and process them.

RabbitMQ message broker is implemented to achieve this requirement. Below is the design flow:

- **Producer service** : This is sample service to produce the event in queue ***https://github.com/adeshpal/event-producer***

- **Audit service:** ***https://github.com/adeshpal/audit_srv***
    - **Event consumer:** Below is the responsibility of audit service
        - This service will continue read the queue and consume the even if any publisher published
        - Process and save the same event into database
    - View Audit APIs:
        - Refer API section for more detail of apis
    - Log Retention
        - Configurable variable is introduced and can tune as per requirement
        - Old logs file will be created as per timestamp and the same would be present in default docker location. (/app/)



Audit Log Rotation: Audit log rotation is essential to manage storage space and ensure that the log size doesn't grow indefinitely. Here's a possible approach:

- Log Rotation Strategy: Added a variable can be configured as per requirement
- Audit log retention policy:
    - Consider archiving old logs to a separate storage (e.g., Amazon S3, Azure Blob Storage) or a different table in the database. This way, you retain the audit trail for compliance purposes while keeping the active logs at a manageable size.
    - Old logs can be removed from the database and moved out in a file that can be later uploaded to aws etc as per requirement.

Microservice Communication: Ensure that all microservices publish relevant events when they make changes to entities. These events should include all the information needed to construct the audit message.

Error Handling: Implement proper error handling and retry mechanisms to deal with any potential failures during the audit message processing or log rotation.

Security Considerations:

Jwt based authentication which will

- Ensure that the audit service itself is secure and accessible only to authorized users.
- Encrypt sensitive data within the audit messages to prevent unauthorized access to sensitive information.

Monitoring and Alerts

Implemented logging for the audit service. In future alerts can be set up to notify administrators in case of any issues or anomalies.

API's

Below are the APIs exposed to full-fill this requirement

API#1: Producer : publish message

Name/Path	POST /producersrv/{version}/message
Description	Message publisher

Requests Parameters	{
	"user_id": 5,
	"service_id" : 3,
	"service_name": "to delete",
	"event_type" : "licDelete",
	"event_details": {
	"oldName":"Google Lic",
	"newName":"Google"
	}
	}

Response	HTTP status: 202
	{
	"status": 1,
	"message": "success",
	"result": {}
	}

API#2: Audit Service

Add User API

Name/Path	POST /auditsrv/{version}/user
Description	Add users in user table JWT based: User_id would be in jwt token

Requests Parameters	{
	“name” : “user1”,
	“email” : “user1@usersrv.com”,
	“Info” : “user 1 of user service”
	}

Response	HTTP 201

Add Admin API

Name/Path	POST /auditsrv/{version}/user/admin
Description	Add admin in user table JWT based: User_id would be in jwt token

Requests Parameters	{
	“name” : “Global Admin”,
	“email” : user1@usersrv.com,
	“role” : 1
	}

Response	{
	“status”: status , #[“success”, “error”]
	“message”: message, #[“Successfully fetched data”, “Failed to get data”, ….. etc]
	“data”: {
	“result”:{}
	}
	}

Error Response	{
	“status”: status , #[“success”, “error”]
	“message”: message, #[“Successfully fetched data”, “Failed to get data”, ….. etc]
	“data”: {
	“result”:{}
	}
	}

Name/Path	GET /auditsrv/{version}/message/{id} JWT: {user_id, }
Description	For users to view all audit messages based in their permission
Query Parameters	user_id

Response	{
	“result”: []
	}

Database Schema

Databases tables:

1. **Role**

id	name	description	permissions
1	Global Admin	Can access audit logs of all services	all
2	Super Admin	Can access audit logs of assigned services	“userSrv, deviceSrv”
3	Service Admin	Can access only respective service logs	

2. **Admin**

id	name	role	email	passord	created_on
1	User 1	1	user1@usersrv.com		
2	User 2	2	user2@usersrv.com		
3	User	3	user2@devicesrv.com		
4	Dummy license	3	user3@licsrv.com		

3. **User**

id	name	email	info	created_on
1	User 1	user1@usersrv.com	"user_1-usrSrv"	
2	User 2	user2@usersrv.com	"user_2-usrSrv"	

3	User 3	user3@devicesrv.com	"user_1-licSrv"	
4	User 4	user4@devicesrv.com	"user_2-licSrv"	
5	User 5	user5@licsrv.com	"user_1-licSrv"	
6	User 6	user6@licsrv.com	"user_2-licSrv"	

4. **Event_info**

id	user_id	event_type	service_id	service_name	event_details	created_on
1	1	usrCreate	1	userSrv	{"oldName": "", "newName": "User 1"}	
2	2	deviceCreate	2	deviceSrv	{"oldName": "", "newName": "iPhone"}	
3	3	licCreate	3	liceSrv	{"oldName": "M365", "newName": "M365 lic"}	

Scalability

Yes

Monitoring and Telemetry

Monitoring and Alerts:

Implemented logging for the audit service. In future alerts can be set up to notify administrators in case of any issues or anomalies.