Case study for Banking Ledger API service

Problem

A *Banking Ledger* is a digital store of assets held by an end-user and all transactional activity (e.g. receiving money, sending money, collecting money on the platform is performed against a Ledger). A single end-user may have more than one Ledger. This ledger is used for banking and the given end-user with the given partner bank in the given currency.

Considerations for software architecture

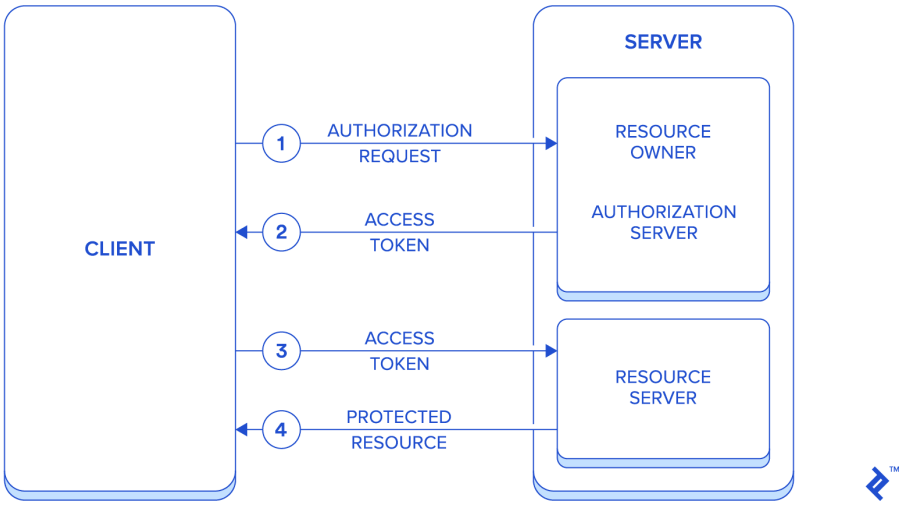
- Security
- Scalability
- Auditability
- Availability

Security

Security is a major consideration for this kind of APIs, because the APIs are dealing with the sensitive date of the end users and with their money transactions. Attackers can easily attack and get end user's sensitive data and funds if we don't implement a good security precautions.

We can use OAuth2 protocol to secure our client server communication and we can overcome our security risk by using this.

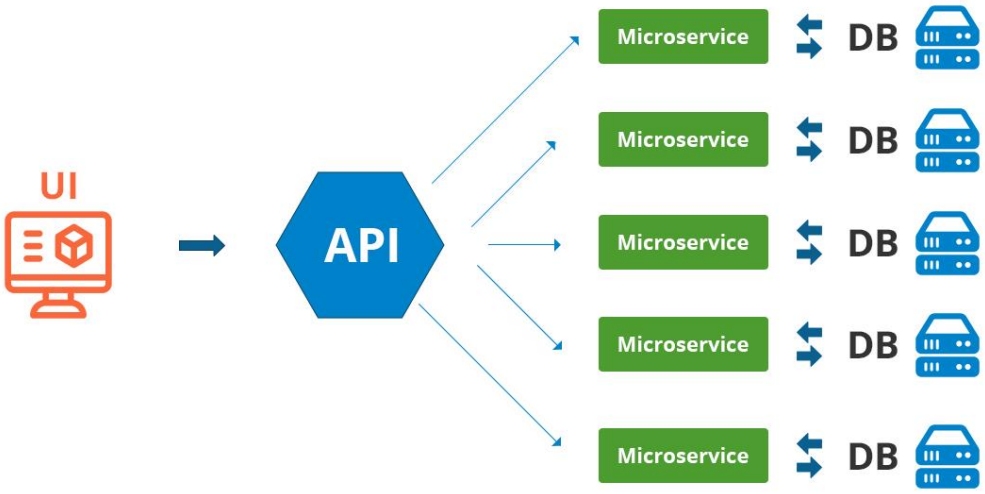
This is how OAuth2 would work with the API server.



Scalability

Scalability will be play a huge role when it comes to performance factor of this kind of API service. When the user count increases, performance will be decreases if we don't have a proper scalable architecture.

We can use a micro services architecture here to overcome this scalability problem. Spring boot or Vert.x will be a good candidate for this. We can introduce multiple instance of same process when we get a higher number of requests as follows.



Auditability

Auditability is playing a huge role in banking industry because each and every transaction should be auditable. Therefore we can't delete any single row from the database even we get the update or delete request from the user.

When the system receives an update query, it is not a good practice to update the database row. Instead of updating the database row, we can keep old record as it is and we can insert a new row. We can handle the old record by introducing a field called 'Record Status'. The latest record's record status field can be set to 0 while the record status field of old record can be set to 1. This way we can easily identify the latest record while maintaining the auditability.

We can introduce 'Delete Flag' field to every database table to keep track of deleted entries. We can keep this field empty for any row and when we get a delete query, can update this field to yes.

We can ensure auditability by introducing above mentioned fields to the application.

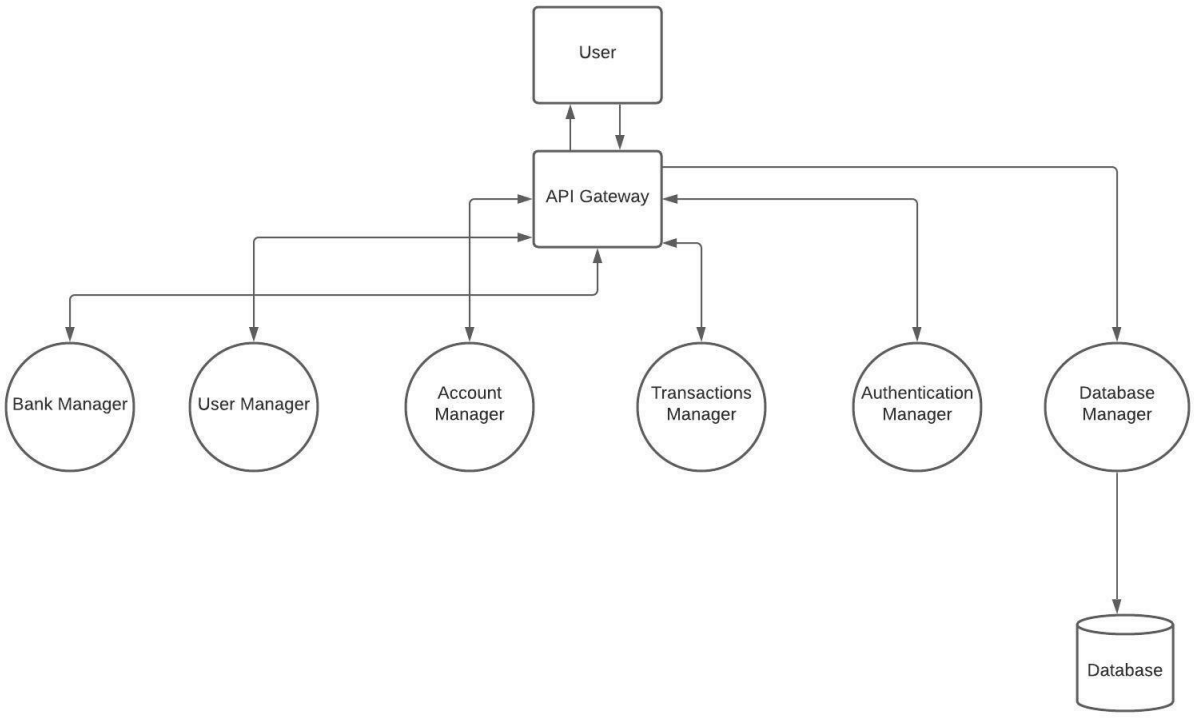
Availability

Another major issue we are facing while developing this kind of sensitive application is availability. We can go for AWS cloud hosting to ensure high availability.

Proposed micro service architecture

We can create separate micro services for different tasks. If we come across with a bottle neck situation, we can introduce another process of the same micro service which the bottleneck occurs as mentioned earlier.

As an example, if we get a lot of request to Transaction Manager micro service, we can introduce another micro service instance of Transaction Manager to cater requests and API Gateway have the capability to handle this.



Proposed Endpoints

Please refer below url to view suggested endpoints. Documented end points are attached in JSON format along with this document (Please refer Appendices). You can use swagger editor to view documented end points from the above json file.

https://akila.stoplight.io/docs/banking-ledger/YXBpOjM4ODAwNjcw-banking-ledger

Technologies

Spring boot for create the API end points

OAuth2 for authentication management

AWS for hosting

Swagger for document the APIs

Oracle database for store the data

Cypress for test automation

Appendices

I attached few documents along with this document in the same git directory that I created to understand the requirement as well as understand the relationships between entities.

- Use case diagram for banking ledger API service.pdf
- ER diagram for banking ledger API service.pdf
- banking-Ledger-API documentation.json