**Service Design**

**Specification**

**TABLE OF CONTENTS**

[1 Overall 2](#_Toc1956872908)

[2 Requirements 2](#_Toc450910951)

[2.1 Features List 2](#_Toc2138046841)

[3 Design 3](#_Toc309391436)

[3.1 User Cases Diagram 3](#_Toc883955465)

[3.1.1 Use Case 3](#_Toc347630309)

[3.2 Service Design 3](#_Toc1467083523)

[3.2.1 Container Architecture 3](#_Toc2013019854)

[3.2.2 Component design 3](#_Toc1367311340)

[3.2.3 Class Design 4](#_Toc179184833)

[3.3 API Design 6](#_Toc787415137)

[4 Specific Considerations 6](#_Toc1291974745)

[4.1 Transactional Messaging 6](#_Toc1012384398)

[4.2 Concurrent Operation 7](#_Toc631642005)

[4.3 CQRS 7](#_Toc995510914)

[4.4 Message Format 8](#_Toc506837821)

[5 Database design 8](#_Toc1503113545)

# Overall

Ledger service handle features includeing account status change, process transaction, update transaction, transaction broadcast, balance change broadcast, history transaction query and balance query, etc.

# Requirements

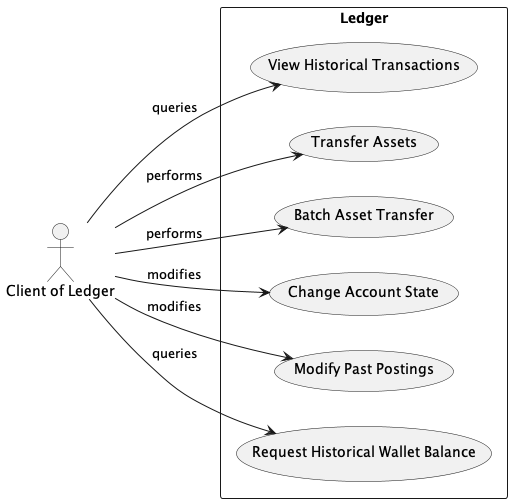
## Features List

|  |  |
| --- | --- |
| **function** | Memo |
| Account status change |  |
| Transaction process | including transaction process, broadcast transaction, broadcast balance change |
| Transaction update | including transaction update, broadcast transaction, broadcast balance change |
| Transaction query |  |
| Balance query | support query balance at specific timestamp |

# Design

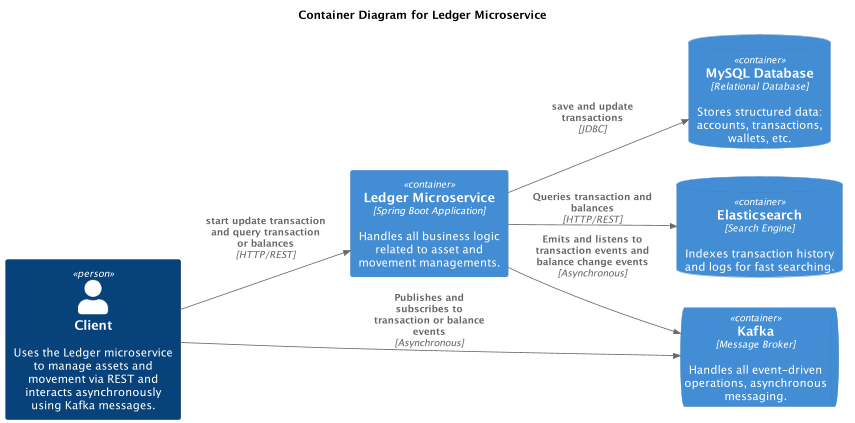
## User Cases Diagram

### Use Case



## Service Design

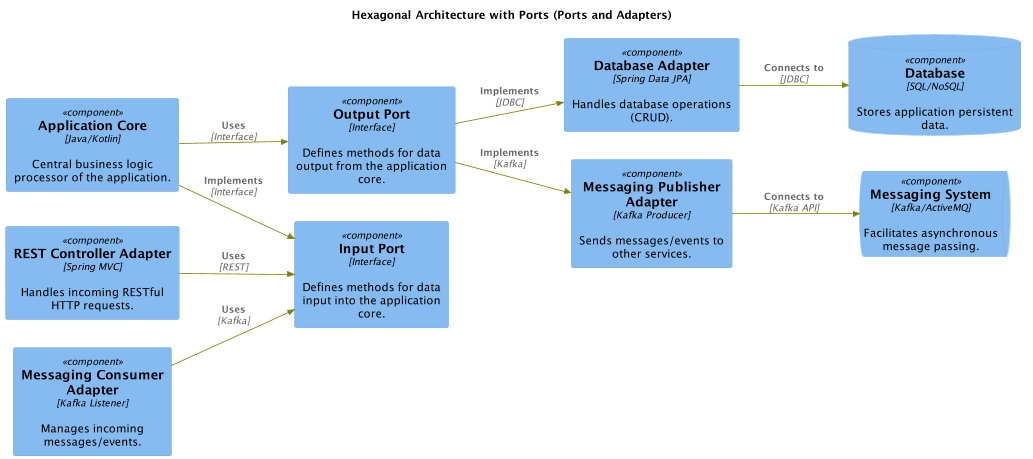
### Container Architecture



### Component design

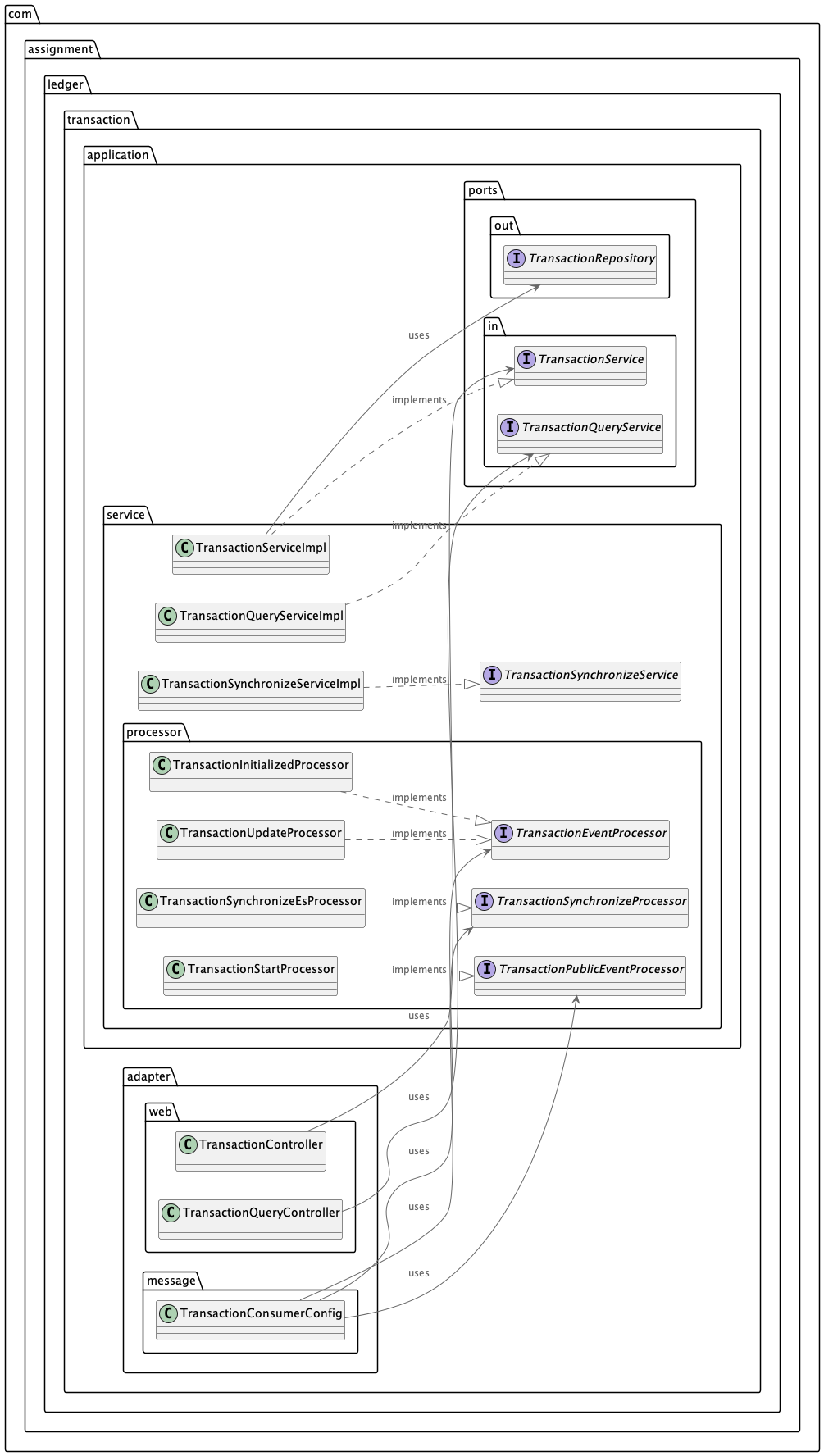
Hexagonal architecture is adopted for clear separation concerns, enhanced maintainablility, scalability, and so on, below is the typical hexagonal architecture.

Hexagonal Architecture is particularly advantageous for applications that require long-term maintenance, need to interact with multiple types of external systems, or anticipate future changes in external interfaces.



### Class Design

Class structure is designed as hexagonal architecture, below is the class diagram of module transaction



* Service package as the applicaton core to provide business logic handling, it implements the in port for the in adapters to call
* In port package provide the application interfaces to outside.
* Out port package provide the outbound interface to downstream component, such as database repository, message producer, etc.
* In adapter call in port to access the application core and let the data flow in the application
* Out adaper is called by application core to persistent data or send event data to interact with outside.

## API Design

Please check the spring openapi doc link <http://${ledger}:8080/swagger-ui/index.html>

The ${ledger} should be replaced by the ip or domain name of running ledger service.

If needed, below attached json format api document could be a reference.

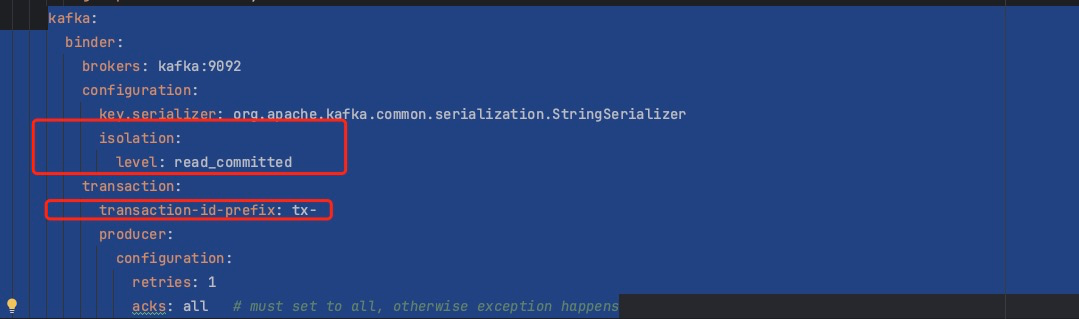


# Specific Considerations

## Transactional Messaging

Nowadays messaging is the very popular communication approach, message could provide high scalability and resilience，but the data consistency is the main challenge for the EDA architecuture.

For the data consistency, ledger service adopts the transactional messaging to guarantee the data consistency between database transaction and message broker i.e. kafka transaction, ‘**all or nothing**’ could be achieved upon this solution, below is the relevant snippet code.



First, we need to config “transaction-id-prefix” to enable kafka transaction

Second, we also need to config “isolation.level” to read\_committed so that consumer can only receive the commtted message, this is important, if consumer **does not config its isolation level properly, it will still receive the uncommitted message** which means transactional message still not work for the data consistency.

Third, the database operations and kafka produce operations should be included in on transaction i.e. in one method which is annotated by annotation @Transactional.

Four, StreamBridge, a component from spring cloud stream should be leveraged to send events so that transactional message could be enabled

So far transactional messaging is a suitable solution for ledger service.

## Concurrent Operation

Ledger service is a final service related to asset transfer, funds transaction, concurrent operation should be considered so that it will not cause any issue under high concurrency pressure.

For balance the database pressure and system performance, optimistic lock solution is adopted to handle this topic.

JPA provide the optimistic lock feature out of box, and it’s applied in ledger service.

One version field will be added in relevant table, and the @version annatation is added to the field in the table’s entity class, so when OptimisticLockException occurs and is caught, the data will not be impacted and further action could be triggered at the first time.

## CQRS

Leger service is a heavy write application , so separate the read and write operations is the proper solution for reduce the database pressure.

Elasticsearch is adpted as the read database as it could provide powerful query capability and high performance and scalability.

Basic idea is when transaction is handled (start, process, update), relevant event will be emitted and a dedicated consumer could listen to the data and synchronize the data to Elasticsearch for query so that read and write could be separated.

## Message Format

A unified message format is very important for integrations among multiple systems.

CloudEvents is a specification for describing event data in a common format, developed by the Cloud Native Computing Foundation (CNCF). It standardizes the way events are represented across services and platforms, enabling interoperable event-driven architectures.

Ledger adopts Cloudevent for enhanced standardization, improved integration and universality.

# **Database design**

Below is the database diagram

