Technical Exercise:

Submitted by: SANDIP REWASKAR

# **Introduction and Design objective:**

Design a payment system for public metro transport. The problem statement is to design the fare calculation engine for TigerCard.

# **System Requirement:**

## **High level Use cases:**

The following are the main functional use cases provided by the application.

* Ticket Management
* Fare calculation
* Search - Train
* Customer’s administration
* Payment Processing
* Reporting
* Analytics

## **Non-functional attributes/requirement**

* **Performance**:

1. The application should support high number of concurrent users, without any performance degradation.
2. The application should be able to achieve low latency.
3. User should be able to search and book tickets with very low latency.

* **Scalability**: The system should be highly scalable.
* **Reliability/Availability requirement**

1. The system must be highly available and should be 99.99% of the time.

* **Maintainability**

1. The system should be developed in such a way that changes can be incorporated easily.
2. Provide low coupling and separation of concerns between components.

* **Extensibility**: It should be easy to add new features or enhance the software

# **Design Solution – Architecture Analysis**

## **Design constraints and assumptions**

* Integration with external payment gateway
* Design will be based on 12 factor app.

## **Design motivation and principles**

* **SOLID**
* **DDD**
* **Reactive manifesto**
* **KISS**
* **YAGNI**
* **12 Factor App**
* **Separation of concerns**
* **Low coupling**
* **Extensibility**: The solution should allow easy addition/extension of new features.
* **Modularity**: The solution should provide intra-layer abstraction by allowing the individual layer components to be independently modified, with minimal impact on components in other layers.
* **Open standards:** During development of integration interfaces and other components, open standards would be followed to prevent vendor lock-in.

## **Use case Realization – High Level Domain Model**

Graphical user interface

Description automatically generated

## **Proposed Architectural Styles:**

The application will be designed based on below architectural styles. The robust design is never limited to a single architectural style but is often a combination of architectural styles that make up the complete system.

|  |  |  |
| --- | --- | --- |
| **Architectural Style** | **Description** | **Advantages** |
| DDD (Domain Driven Design) | Domain model based on underlying business domain | Common domain model  Evolving architecture |
| Reactive | Services that are flexible, scalable, and highly responsive | resilient services |
| Microservices | Small services modelled around bounded context. | Manageable small services which are much faster to develop, and much easier to understand and maintain. |
| Component based architecture. | Reusable functional or logical components that expose well-defined communication interfaces. | Reusability  Clean interfaces |
| Event Driven architecture | Using events to implement business transactions that span multiple services | Decoupled design  Better scalability and fault tolerant architecture. |
| Polyglot persistence | Different data storage technologies to handle different data storage | extensibility |
| Event sourcing | Persisting each state of an event. | Reactive approach.  Enables concurrent, distributed systems to achieve high performance, scalability, and resilience. |

## **Architectural Patterns and proposed solutions:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Problem | Patterns | Solution | Alternative | Reason |
| How to design resilient services | Reactive manifesto | Reactive Microservices | Synchronous services | Services that are flexible, scalable, and highly responsive |
| How to reliably/atomically update the database and publish messages/events | Event Sourcing and CQRS | Event sourcing persists the state of a business entity as a sequence of state-changing events. | Event driven |  |
| How to implement an event driven architecture | Event bus | Kafka | Spring cloud |  |
| How to handle distributed transaction and database consistency | Event Sourcing  Saga Pattern  CQRS |  |  | Eventual consistency |
| How to implement autoscaling of services | Containerization | Kubernetes | Docker Swarm | Ability to scale up and down |
| Automated infra provisioning | Containerization | Docker/Kubernetes |  |  |
| Monitoring and distributed tracing. | Distributed tracing | Dynatrace | AppDynamics |  |
| High availability | Active- Active topology | Global load balancer  and data mirroring |  |  |
| Reference data | Read through and write through | Redis Cache |  |  |

# **High level Design solution:**

## **Proposed Solution:**

## 

Graphical user interface, diagram

Description automatically generated