**PROJECT REPORT**

(Project Term January-May 2024)

## TICKTING APPLICATION

Submitted by

**Shubhansu Kumar Singh**

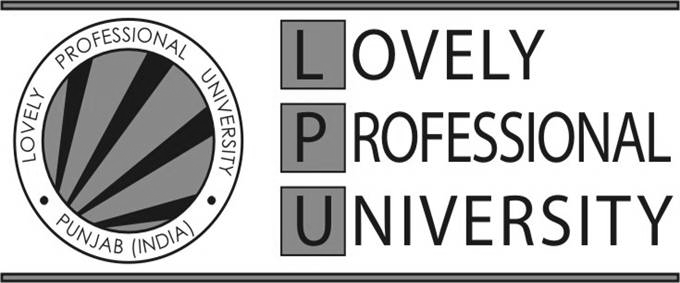
**12104991**

**INT332**

Under the Guidance of

**Dr. Harpreet Kaur**

# School of Computer Science and Engineering



**PAC FormDECLARATION**

We hereby declare that the project work entitled “Ticketing application” is an authentic record of our own work carried out as requirements of Capstone Project for the award of B. Tech degree in Computer Science and engineering from Lovely Professional University, Phagwara, under the guidance of Dr. Harpreet Kaur, during January to May 2024. All the information furnished in this project report is based on our own intensive work and is genuine.

Shubhansu Kumar Singh

12104991

**CERTIFICATE**

This is to certify that the declaration statement made by this group of students is correct to the best of my knowledge and belief. They have completed this Project under my guidance and supervision. The present work is the result of their original investigation, effort and study. No part of the work has ever been submitted for any other degree at any University. Project is fit for the submission and partial fulfillment of the conditions for the award of B.Tech degree in Computer Science and Engineering from Lovely Professional University, Phagwara.

**Signature and Name of the Mentor**

**Designation**

**School of Computer Science and Engineering,**

Lovely Professional University,

Phagwara, Punjab.

Date :

**ACKNOWLEDGEMENT**

**TABLE OF CONTENTS**

Inner first page……………………………………………………………………….(i) PAC form……………………………………………………………………………(ii)

Declaration…...……………………………………………………………………..(iii)

Certificate……..…….………………………………………………………………(iv)

Acknowledgement…………………...………………………………………….......(v)

Table of Contents……………………...……………………………………………(vi)

**1. INTRODUCTION 1**

## 1.1. SECOND-LEVEL SUBHEADING 1

## 1.2. ANOTHER SECOND-LEVEL SUBHEADING 1

## 1.2.1. THIRD-LEVEL SUBHEADING 1

**1. INTRODUCTION**

Microservices architecture is an approach to software development where a large application is decomposed into smaller, independent services, each responsible for a specific business function. These services are loosely coupled and communicate with each other through well-defined APIs.

### 1.1. WHAT ARE MICROSERVICES?

Microservices break down a monolithic application into smaller, self-contained services, each running its own process and communicating with lightweight mechanisms like HTTP or messaging queues. Each microservice is developed, deployed, and scaled independently, allowing teams to work on different services simultaneously and deploy updates without affecting the entire application.

## 1.2. WHY MICROSERVICE IN TICKETING APPLICATION

1. **Decomposition:** In a ticketing application, various functionalities such as authentication, ticket ordering, payment processing, and expiration handling can be broken down into separate microservices. Each microservice focuses on one aspect of the application's functionality.
2. **Independence:** Microservices are independently deployable and scalable. This means that changes or updates to one microservice do not require redeployment of the entire application. For example, if there's an update to the authentication service, it can be deployed without affecting other parts of the application.
3. **Technology Diversity:** Microservices allow for the use of different technologies and programming languages suited to the specific needs of each service. For instance, the authentication service might be written in Node.js, while the payment service might be written in Java.
4. **Scalability:** Each microservice can be scaled independently based on demand. For instance, during peak ticket sales, the orders and payments services can be scaled up to handle increased traffic, while other services remain unaffected.
5. **Resilience:** Microservices architecture improves fault isolation. If one microservice fails, it does not bring down the entire application. Failures are contained within the affected microservice, and other services continue to function.
6. **Ease of Maintenance:** Microservices promote modularization, making it easier to understand and maintain the codebase. Developers can focus on a single service's logic without needing to understand the entire application.
7. **API Gateway:** An API gateway can be used to provide a single entry point for clients to access various microservices. It handles routing, authentication, and other cross-cutting concerns, simplifying client interactions with the microservices architecture.
8. **Database per Service:** Each microservice can have its own database, allowing for better isolation of data and reducing the risk of data corruption. For example, the orders service might use a relational database for storing order information, while the authentication service might use a NoSQL database for user data.
9. **Event-Driven Architecture:** Microservices can communicate asynchronously through events. For instance, when a new order is placed, an event can be published to a message broker like NATS, which other services can subscribe to and act upon, ensuring loose coupling between services.
10. **DevOps Practices:** Microservices encourage DevOps practices like continuous integration, continuous deployment, and infrastructure as code. Automation tools and container orchestration platforms like Kubernetes are often used to manage and deploy microservices at scale.

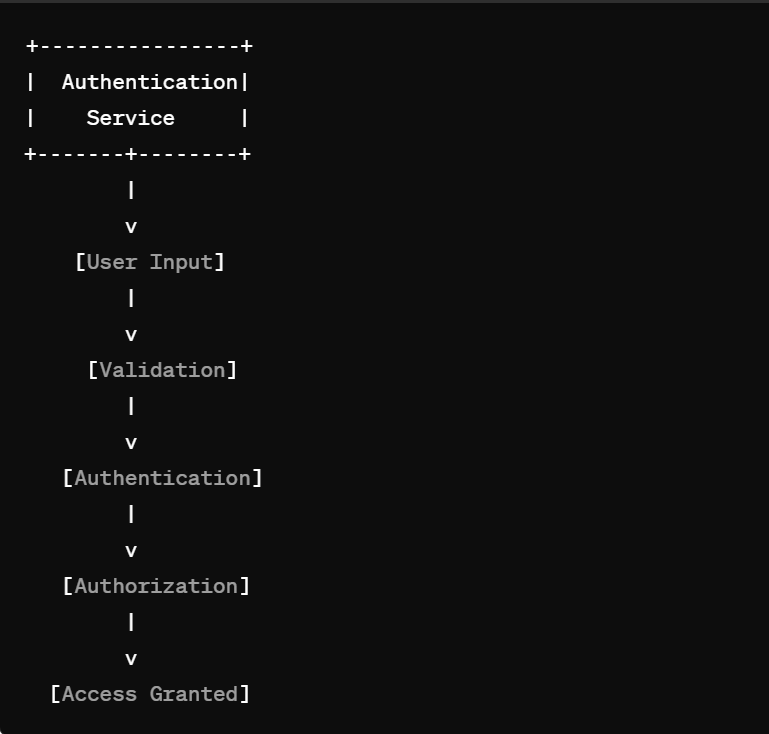
**2. MICROSERVICES USED**

The Microservices used to create this Ticketing application :

1. Authentication Service
2. Orders Service
3. Payments Service
4. Expiration Service
5. Client Service (User Interface)
6. NATS (Messaging Service)

### 2.1. Authentication Service:

* **Description:** This microservice handles user authentication and authorization.
* **Diagram:**

****

* **Flowchart:**

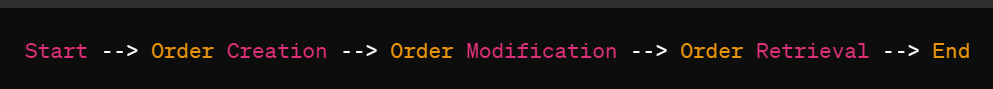


### 2.2. Orders Service:

* **Description:** Manages the creation, retrieval, and modification of ticket orders.
* **Diagram:**

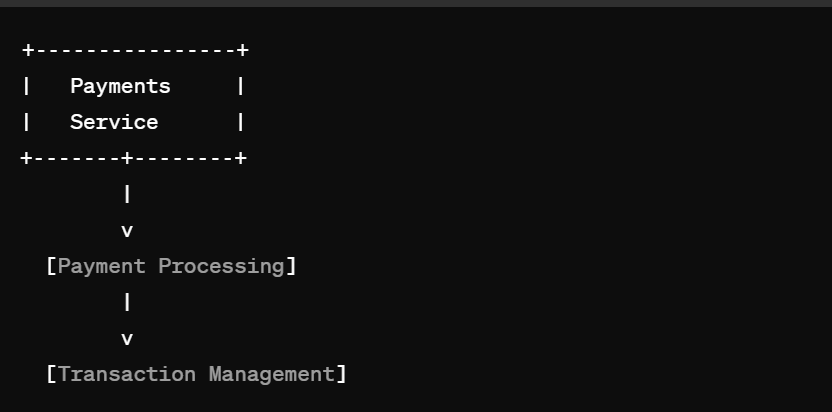
****

* **Flowchart:**

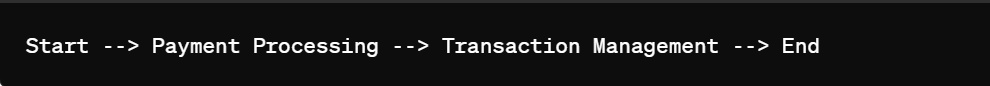
****

### 2.3. Payments Service:

* **Description:** Handles payment processing and transaction management.
* **Diagram:**

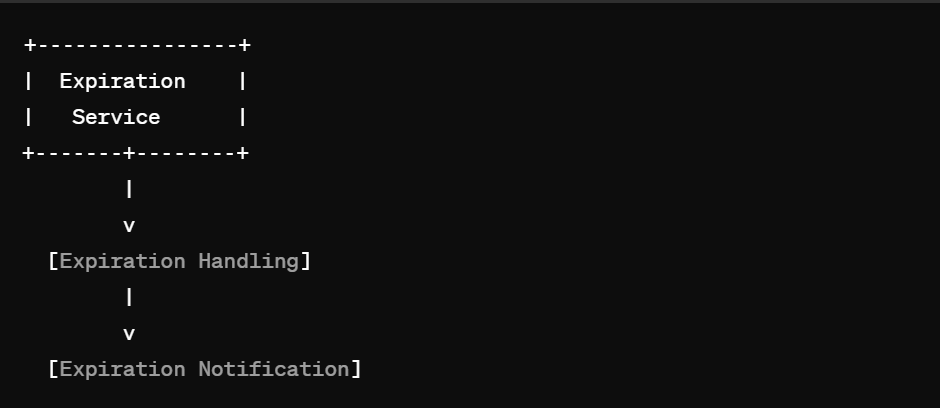
****

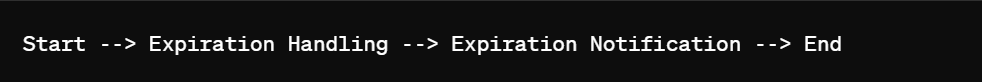
* **Flowchart:**

****

### 2.4. Expiration Service:

* **Description:** Manages ticket expiration and related functionalities.
* **Diagram:**

****

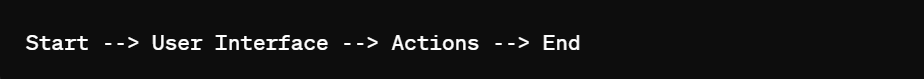
* **Flowchart:**

### 2.5. Client Service (User Interface):

* **Description:** Provides the user interface for interacting with the ticketing system.
* **Diagram:**

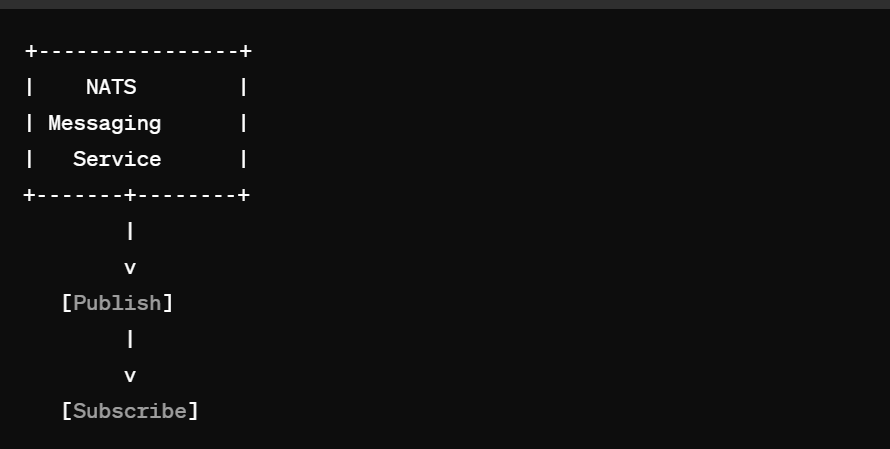
****

* **Flowchart:**

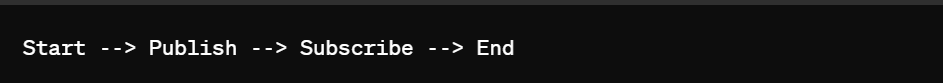
****

### 2.6. NATS (Messaging Service):

* **Description:** Facilitates asynchronous messaging between different microservices.
* **Diagram:**

****

* **Flowchart:**

****

These diagrams and flowcharts provide a simplified visualization of the interactions and functionalities of each microservice within the ticketing application.

**3. DEPLOYMENT**

In deploying this ticketing application via Kubernetes, several microservices are orchestrated. MongoDB handles data storage, NATS facilitates messaging, while services for authentication, orders, payments, and expiration are deployed. Each microservice is encapsulated within a Deployment, specifying container configurations and environment variables. These variables establish connections to essential services such as MongoDB, NATS, and Redis, ensuring seamless communication and functionality across the application. Kubernetes manages the scalability and resilience of these deployments, ensuring efficient resource utilization and fault tolerance, thus providing a robust foundation for the ticketing application's operation.

**3.1. Deployment (tickets-mongo-depl.yaml):**

Kubernetes Deployment to manage MongoDB Pods and a Service to expose MongoDB to other components of your ticketing application. The Deployment ensures that one MongoDB Pod is running, and the Service enables communication with MongoDB Pods using a stable endpoint.

1. Top of Form

yaapiVersion: apps/v1

kind: Deployment

metadata:

  name: tickets-mongo-depl

spec:

  replicas: 1

  selector:

    matchLabels:

      app: tickets-mongo

  template:

    metadata:

      labels:

        app: tickets-mongo

    spec:

      containers:

        - name: tickets-mongo

          image: mongo

* **apiVersion**: Specifies the Kubernetes API version being used for this resource.
* **kind**: Defines the type of Kubernetes resource, in this case, a Deployment.
* **metadata**: Provides metadata for the Deployment, including its name.
* **spec:** Describes the desired state for the Deployment.
* **replicas**: Specifies the number of replica Pods to create, in this case, 1 replica.
* **selector**: Defines how the Deployment selects which Pods to manage. matchLabels: Specifies the labels used for selecting Pods, matching with Pods' labels.
* **template**: Defines the Pod template used to create new Pods.
* **metadata:** Labels applied to Pods created from this template.
* **spec**: Defines the specification for the Pod.
* **containers**: Specifies the containers to run within the Pod.
* **name**: Name of the container.
* **image**: Docker image to use for the container, in this case, the official MongoDB image 1

**3.2 Service (tickets-mongo-srv.yaml):**

YapiVersion: v1

kind: Service

metadata:

  name: tickets-mongo-srv

spec:

  selector:

    app: tickets-mongo

  ports:

    - name: db

      protocol: TCP

      port: 27017

      targetPort: 27017

* **app:** Label selector matching Pods with the label app: tickets-mongo.
* **ports**: Specifies the ports that the Service exposes.
* **name**: Name of the port.
* **protocol**: Protocol used by the port (TCP in this case).
* **port:** Port number exposed by the Service.
* **targetPort**: Port number on the Pods to forward traffic to (matches MongoDB's default port 27017).

**3.3** **Deployment (tickets-depl.yaml):**

This Deployment is for the main component of your ticketing application (**tickets**), pulling its Docker image from **rallycoding/tickets**. It sets several environment variables necessary for the application to function, including configuration for connecting to NATS messaging server (**NATS\_CLIENT\_ID**, **NATS\_URL**, **NATS\_CLUSTER\_ID**) and MongoDB (**MONGO\_URI**), as well as JWT secret key (**JWT\_KEY**) retrieved from a Kubernetes Secret named **jwt-secret**.

y apiVersion: apps/v1

kind: Deployment

metadata:

  name: tickets-depl

spec:

  replicas: 1

  selector:

    matchLabels:

      app: tickets

  template:

    metadata:

      labels:

        app: tickets

    spec:

      containers:

        - name: tickets

          image: rallycoding/tickets

          env:

            - name: NATS\_CLIENT\_ID

              valueFrom:

                fieldRef:

                  fieldPath: metadata.name

            - name: NATS\_URL

              value: 'http://nats-srv:4222'

            - name: NATS\_CLUSTER\_ID

              value: ticketing

            - name: MONGO\_URI

              value: 'mongodb://tickets-mongo-srv:27017/tickets'

            - name: JWT\_KEY

              valueFrom:

                secretKeyRef:

                  name: jwt-secret

                  key: JWT\_KEY

* **replicas**: Specifies the number of replica Pods to create, in this case, 1 replica.
* **selector**: Defines how the Deployment selects which Pods to manage.
* **matchLabels**: Specifies the labels used for selecting Pods, matching with Pods' labels.
* **template**: Defines the Pod template used to create new Pods.
* **metadata**: Labels applied to Pods created from this template.
* **spec**: Defines the specification for the Pod.
* **containers**: Specifies the containers to run within the Pod.
* **name**: Name of the container.
* **image**: Docker image to use for the container (**rallycoding/tickets** in this case).
* **env**: Defines environment variables for the container.
* **name**: Name of the environment variable.
* **value** or **valueFrom**: Value of the environment variable, either directly specified or retrieved from another source.
* **fieldRef**: Retrieves the value from a field in the Kubernetes object (in this case, the Pod's metadata name).
* **secretKeyRef**: Retrieves the value from a Kubernetes Secret (**jwt-secret** in this case).

### 3.4 ****Deployment (auth-mongo-depl.yaml):****

This Deployment ensures that one instance of MongoDB is running, with the Pod labeled as **app: auth-mongo**. The MongoDB image (**mongo**) is used for this container, indicating that this Deployment is setting up a MongoDB database instance specifically for authentication purposes within your ticketing application.

apiVersion: apps/v1

kind: Deployment

metadata:

  name: auth-mongo-depl

spec:

  replicas: 1

  selector:

    matchLabels:

      app: auth-mongo

  template:

    metadata:

      labels:

        app: auth-mongo

    spec:

      containers:

        - name: auth-mongo

          image: mongo

* **matchLabels**: Specifies the labels used for selecting Pods, matching with Pods' labels.
* **template**: Defines the Pod template used to create new Pods.
* **metadata**: Labels applied to Pods created from this template.
* **spec**: Defines the specification for the Pod.
* **containers**: Specifies the containers to run within the Pod.
* **name**: Name of the container.
* **image**: Docker image to use for the container, in this case, the official MongoDB image.

### 3.5 ****Deployment (auth-depl.yaml):****

This Deployment sets up the authentication component (**auth**) using the Docker image **rallycoding/auth**. It configures environment variables required for the authentication service, such as **MONGO\_URI** for connecting to the MongoDB database (**mongodb://auth-mongo-srv:27017/auth**) and **JWT\_KEY** for JWT token encryption, retrieved from a Kubernetes Secret named **jwt-secret**

apiVersion: apps/v1

kind: Deployment

metadata:

  name: auth-depl

spec:

  replicas: 1

  selector:

    matchLabels:

      app: auth

  template:

    metadata:

      labels:

        app: auth

    spec:

      containers:

        - name: auth

          image: rallycoding/auth

          env:

            - name: MONGO\_URI

              value: 'mongodb://auth-mongo-srv:27017/auth'

            - name: JWT\_KEY

              valueFrom:

                secretKeyRef:

                  name: jwt-secret

                  key: JWT\_KEY

* **containers**: Specifies the containers to run within the Pod.
* **name**: Name of the container.
* **image**: Docker image to use for the container (**rallycoding/auth** in this case).
* **env**: Defines environment variables for the container.
* **name**: Name of the environment variable.
* **value**: Value of the environment variable.
* **valueFrom**: Specifies that the value of the environment variable is retrieved from another source.
* **secretKeyRef**: Retrieves the value from a Kubernetes Secret (**jwt-secret** in this case).

**3.6 Deployment (nats-depl.yaml):**

Kubernetes Deployment and Service for setting up a NATS (a messaging system) instance within your ticketing application. This Deployment sets up a single instance of NATS with the specified arguments. The image used is **nats-streaming:0.17.0**, and the arguments configure NATS with certain parameters (**-p**, **-m**, **-hbi**, **-hbt**, **-hbf**, **-SD**, **-cid**). The Deployment ensures that one instance of NATS is running, and the Service exposes ports for communication with the NATS instance.

Top of Form

apiVersion: apps/v1

kind: Deployment

metadata:

  name: nats-depl

spec:

  replicas: 1

  selector:

    matchLabels:

      app: nats

  template:

    metadata:

      labels:

        app: nats

    spec:

      containers:

        - name: nats

          image: nats-streaming:0.17.0

          args:

            [

              '-p',

              '4222',

              '-m',

              '8222',

              '-hbi',

              '5s',

              '-hbt',

              '5s',

              '-hbf',

              '2',

              '-SD',

              '-cid',

              'ticketing',

            ]

* **spec**: Defines the specification for the Pod.
* **containers**: Specifies the containers to run within the Pod.
* **name**: Name of the container.
* **image**: Docker image to use for the container (**nats-streaming:0.17.0** in this case).
* **args**: Additional command-line arguments passed to the NATS container.

apiVersion: v1

kind: Service

metadata:

  name: nats-srv

spec:

  selector:

    app: nats

  ports:

    - name: client

      protocol: TCP

      port: 4222

      targetPort: 4222

    - name: monitoring

      protocol: TCP

      port: 8222

      targetPort: 8222

* **ports**: Specifies the ports that the Service exposes.
* **name**: Name of the port.
* **protocol**: Protocol used by the port (TCP in this case).
* **port**: Port number exposed by the Service.
* **targetPort**: Port number on the Pods to forward traffic to.

This Service exposes two ports, **4222** and **8222**, for clients and monitoring purposes, respectively. It routes traffic to Pods labeled with **app: nats**.

**3.7 Deployment (client-depl.yaml):**

Deployment for the client component of your ticketing application. This Deployment ensures that one instance of the client component is running, using the Docker image **rallycoding/ticketing-client**. The client component likely serves the user interface or frontend of your ticketing application.

apiVersion: apps/v1

kind: Deployment

metadata:

  name: client-depl

spec:

  replicas: 1

  selector:

    matchLabels:

      app: client

  template:

    metadata:

      labels:

        app: client

    spec:

      containers:

        - name: client

          image: rallycoding/ticketing-client

* **containers**: Specifies the containers to run within the Pod.
* **name**: Name of the container.
* **image**: Docker image to use for the container (**rallycoding/ticketing-client** in this case).

**3.8 Deployment (orders-depl.yaml):**

Deployment for the orders component of your ticketing application. This Deployment sets up the orders component using the Docker image **rallycoding/orders**. It configures environment variables required for the orders service, such as **NATS\_CLIENT\_ID**, **NATS\_URL**, **NATS\_CLUSTER\_ID** for NATS messaging, **MONGO\_URI** for connecting to the MongoDB database, and **JWT\_KEY** for JWT token encryption, retrieved from a Kubernetes Secret named **jwt-secret**.

The **NATS\_CLIENT\_ID** is set to the name of the Pod (retrieved from the metadata field) to ensure uniqueness when connecting to the NATS messaging system. The **NATS\_URL** points to the NATS server (**nats-srv**) running within the Kubernetes cluster.

apiVersion: apps/v1

kind: Deployment

metadata:

  name: orders-depl

spec:

  replicas: 1

  selector:

    matchLabels:

      app: orders

  template:

    metadata:

      labels:

        app: orders

    spec:

      containers:

        - name: orders

          image: rallycoding/orders

          env:

            - name: NATS\_CLIENT\_ID

              valueFrom:

                fieldRef:

                  fieldPath: metadata.name

            - name: NATS\_URL

              value: 'http://nats-srv:4222'

            - name: NATS\_CLUSTER\_ID

              value: ticketing

            - name: MONGO\_URI

              value: 'mongodb://orders-mongo-srv:27017/orders'

            - name: JWT\_KEY

              valueFrom:

                secretKeyRef:

                  name: jwt-secret

                  key: JWT\_KEY

* **image**: Docker image to use for the container (**rallycoding/orders** in this case).
* **env**: Defines environment variables for the container.
* **name**: Name of the environment variable.
* **value** or **valueFrom**: Value of the environment variable, either directly specified or retrieved from another source.
* **fieldRef**: Retrieves the value from a field in the Kubernetes object (in this case, the Pod's metadata name).
* **secretKeyRef**: Retrieves the value from a Kubernetes Secret (**jwt-secret** in this case).

**3.9 Deployment (payments-depl.yaml):**

Deployment for the payment’s component of your ticketing application. This Deployment sets up the payment’s component using the Docker image **rallycoding/payments**. It configures environment variables required for the payments service, such as **NATS\_CLIENT\_ID**, **NATS\_URL**, **NATS\_CLUSTER\_ID** for NATS messaging, **MONGO\_URI** for connecting to the MongoDB database, **JWT\_KEY** for JWT token encryption (retrieved from a Kubernetes Secret named **jwt-secret**), and **STRIPE\_KEY** for Stripe API integration (retrieved from a Kubernetes Secret named **stripe-secret**).

apiVersion: apps/v1

kind: Deployment

metadata:

  name: payments-depl

spec:

  replicas: 1

  selector:

    matchLabels:

      app: payments

  template:

    metadata:

      labels:

        app: payments

    spec:

      containers:

        - name: payments

          image: rallycoding/payments

          env:

            - name: NATS\_CLIENT\_ID

              valueFrom:

                fieldRef:

                  fieldPath: metadata.name

            - name: NATS\_URL

              value: 'http://nats-srv:4222'

            - name: NATS\_CLUSTER\_ID

              value: ticketing

            - name: MONGO\_URI

              value: 'mongodb://payments-mongo-srv:27017/payments'

            - name: JWT\_KEY

              valueFrom:

                secretKeyRef:

                  name: jwt-secret

                  key: JWT\_KEY

            - name: STRIPE\_KEY

              valueFrom:

                secretKeyRef:

                  name: stripe-secret

                  key: STRIPE\_KEY

* **name**: Name of the container.
* **image**: Docker image to use for the container, which is **rallycoding/payments** in this case.
* **env**: Defines environment variables for the container.
* **name**: Name of the environment variable.
* **valueFrom**: Specifies that the value of the environment variable is retrieved from another source.
* **fieldRef**: Retrieves the value from a field in the Kubernetes object (in this case, the Pod's metadata name).
* **secretKeyRef**: Retrieves the value from a Kubernetes Secret (such as **jwt-secret** and **stripe-secret**).

**3.10 Deployment (expiration-depl.yaml):**

Deployment for the expiration component of your ticketing application. This Deployment sets up the expiration component using the Docker image **rallycoding/expiration**. It configures environment variables required for the expiration service, such as **NATS\_CLIENT\_ID**, **NATS\_URL**, **NATS\_CLUSTER\_ID** for NATS messaging, and **REDIS\_HOST** for connecting to the Redis database.

apiVersion: apps/v1

kind: Deployment

metadata:

  name: expiration-depl

spec:

  replicas: 1

  selector:

    matchLabels:

      app: expiration

  template:

    metadata:

      labels:

        app: expiration

    spec:

      containers:

        - name: expiration

          image: rallycoding/expiration

          env:

            - name: NATS\_CLIENT\_ID

              valueFrom:

                fieldRef:

                  fieldPath: metadata.name

            - name: NATS\_URL

              value: 'http://nats-srv:4222'

            - name: NATS\_CLUSTER\_ID

              value: ticketing

            - name: REDIS\_HOST

              value: expiration-redis-srv

* **name**: Name of the container.
* **image**: Docker image to use for the container, which is **rallycoding/expiration** in this case.
* **env**: Defines environment variables for the container.
* **name**: Name of the environment variable.
* **valueFrom**: Specifies that the value of the environment variable is retrieved from another source.
* **fieldRef**: Retrieves the value from a field in the Kubernetes object (in this case, the Pod's metadata name).

**4. Data Flow Diagram (DFD)**

Illustrating the flow of data and interactions between different components in the ticketing application:



In this diagram:

* Users interact with the User Interface to perform actions like purchasing tickets.
* The User Interface communicates with the Authentication service to authenticate users.
* Orders are handled by the Orders service, which manages ticket orders.
* Payments are processed by the Payments service, handling payment transactions.
* Expiration service manages ticket expiration and related functionalities.
* MongoDB is used for data storage, including user information, orders, and ticket details.
* NATS facilitates messaging between different components, ensuring seamless communication.
* Redis is used for caching and other data storage needs.

This DFD provides a high-level overview of how data flows through various components of the ticketing application.

**5. GITHUB LINK**

<https://github.com/shubhansu-kr/GitTix>