Chatting Application Using Apache Kafka

School of Engineering & Computer Sciences
Texas A & M University, Corpus Christi

Mr. Yongzhi Wang

Advanced Operating Systems Project Report

I. Abstract

Our objective is to use Apache Kafka to develop and implement a real-time messaging system with an emphasis on its distributed system features. Making the system able to process a lot of messages while keeping a respectable response time is the goal. Setting up Zookeeper, Kafka, and defining the system architecture are all part of the development process. Spring Boot, Zookeeper, ReactJS, and Apache Kafka are the tools and technologies we employed. The chat program made use of Kafka's publish-subscribe messaging structure. The architecture and design of the chat program are also covered in this study, along with how Kafka's features—like message replication—can support fault tolerance and data consistency.

II. Introduction

Chatting applications are very important and are widespread all over the world for individuals to communicate with each other in real time irrespective of their geographical location. Because users expect their messages to be delivered instantly, real-time messaging is therefore an essential component of any chat service. Creating a restful API enables clients to submit HTTP requests to the server in order to deliver or receive messages, which is one way to create chat applications. However, clients won't be informed when new messages are accessible, which is a significant drawback of this system. An additional option is to establish connections between clients and servers through the WebSocket protocol.

Creating chat systems with Apache Kafka can help solve this issue. When compared to most of the messaging systems, Kafka gives higher throughput, built-in splitting, replication, consistency, and fault tolerance which makes it a good choice for message processing. Because of this, Apache Kafka functions well as a substitute for more traditional message brokers like ActiveMQ and RabbitMQ.

When it comes to accepting, delivering, and routing messages between two or more systems, a message broker serves as the focal point. One well-known message processing program that makes use of Apache Kafka is Twitter to process user interactions, tweets, and other data related issues in real-time.

The goal of this project is to create real-time chat messaging using Apache Kafka. Messages are not lost even in the case of a failure thanks to Kafka's scalability and availability features. The WebSocket protocol is used in this chat application to enable full-duplex, real-time user communication. The suggested method guarantees message delivery without requiring polling. This resolution offers a strong platform on which to build a highly dependable real-time chat application that delivers messages in a reasonable amount of time. An efficient method for creating a scalable and long-lasting messaging system is Apache Kafka.

III. CONTRIBUTION

Team Members:

Yashashwini Devineni (A04315509) - Backend, contribution - 25%

Bindu Naga Varshini Cherukupalli (A04313673) - Frontend, contribution - 25%

Kiranmayee Goud Panthangi (A04324454) - Kafka, contribution - 25%

Varshithai Kesa (A04319979) - Database, contribution - 25%

IV. BACKGROUND AND OBJECTIVE

A. History of chatting application

The earliest specialised online chat service was perhaps CompuServe's CB Simulator, which debuted in 1980 (Barot & Oren 2015, 21). However, new chat apps were not more generally available to the public until the 1990s, with the increased usage of the internet and personal pc's. Chatting applications like AOL Instant Messenger, and MSN Messenger became more and more popular and contributed to the rise in popularity of instant messaging as a communication method, claim Barot and Oren (2015, 21).

As social media platforms gained popularity in the early 2000s, businesses began creating new chatting applications that could be included into these platforms. For example, when Facebook introduced its chatting features in 2008, users could interact with each other in real time while connected into their accounts on the social network. The public began using chat programs more frequently in the late 2000s and early 2010s because of the development of smartphones and mobile technologies (Inno Instant 2022). For instance, when WhatsApp was introduced in 2009, it rose to prominence as one of the most widely used chatting programs worldwide. 2.2 billion people were using WhatsApp as of 2021 (Iqbal, 2023).

Chat programs are becoming a necessary component of contemporary communication. They can convey images and location data in addition to messages. Additionally, users can now make audio or video calls using chatting software. As new features and technologies are added, the chat software keeps changing to better serve their users and increase their security and dependability.

B. Project background and purpose

In recent times there has been a rise in the use of chat programs since they offer a simple and practical means of rapid communication. A software program that allowed users to send and receive text messages over the internet was the first definition of an online chat application (Tarud

2021). Over time, chat apps have developed with the internet, moving from simple text-based messaging systems to more complex ones that enable multimedia content and cutting-edge capabilities like file sharing, group messaging, and audio and video chats.

The popularity of chat programs has increased because of the proliferation of smartphones and mobile computing, which allow users to access their chat apps from any location. Consequently, chat programs are extensively employed for diverse objectives such as interpersonal connection among friends and family, business-to-business exchanges, and customer support between enterprises and clients.

Since chat apps make it possible for distant teams to efficiently interact and work together, they have also become essential in the digital workplace. The COVID-19 epidemic has made many firms adopt remote work, which has led to an even greater adoption of chat software.

However, there are several obstacles to overcome when developing a chat application that can manage heavy traffic and massive amounts of data in real time. When building and executing a chat application, developers must take performance, scalability, and reliability into account. To solve these issues, several messaging solutions are available, such as Apache Kafka and RabbitMQ. Although both technologies are free and open-source messaging platforms that can be used for chat apps, they differ in significant ways that could make them better suited for particular use cases.

RabbitMQ facilitates many messaging patterns, such as topic exchange, fanout exchange, and direct exchange, whereas Kafka employs a publish-subscribe messaging style. It might be more appropriate to use the publish-subscribe structure of Kafka in a chat application where users must subscribe to chat channels and receive messages instantly. RabbitMQ may have a little bit more delay than Kafka, even though it is still quick and dependable. This is because of its more intricate architecture. Kafka is a perfect fit for chat applications that need to process a lot of messages in real time and at a fast throughput. It can also manage massive amounts of data and has outstanding scalability.

RabbitMQ and Apache Kafka can both be used with chat programs, however if real-time message processing, high throughput, and low latency are needed, Kafka would be a better fit. However, if a more intricate messaging pattern is required, RabbitMQ might be a better option. Consequently, Apache Kafka was selected as the messaging platform for the suggested real-time chat application following thorough examination and analysis.

ReactJS and Spring Boot were chosen for the planned chat application's development. Real-time message processing was made easier by Spring Boot's event-driven design, and the front-end development of the implemented application was made simpler by ReactJS. To create the chat application, however, a variety of frameworks or technologies could be combined with Apache Kafka. Kafka is compatible with numerous programming languages, including Go, Python, C/C++, and many more.

C. Challenges and solutions

To ensure the suggested chat application's success, several issues must be resolved. Among the main obstacles is real-time messaging. Instantaneous delivery of messages is expected. This calls for a reliable messaging platform that can manage user conversation in real time. Data delivery via the HTTP protocol will be delayed because HTTP must complete a full cycle, which includes a request from the client and a response from the server to the matching request (Figure 1). Furthermore, for the client to receive messages via an HTTP connection, as seen in Figure 1, the client must send a request to the server.

WebSocket protocol can be utilized to enable real-time communication between users, which would solve this issue. When using WebSocket, the client sends a request to the server and waits for a response before establishing a connection with the server (Figure 1). Users can get messages quickly from the chat application and polling can be avoided by using the WebSocket protocol.

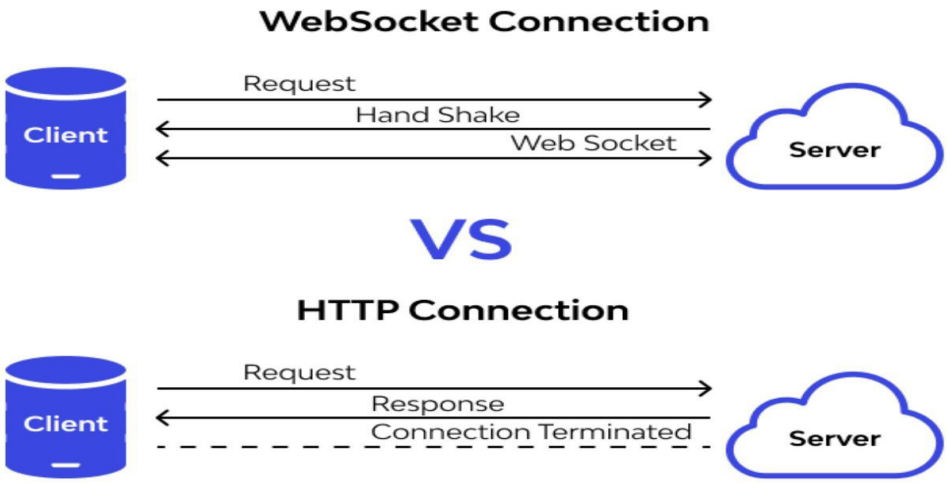


FIGURE 1. WebSocket protocol versus HTTP protocol.

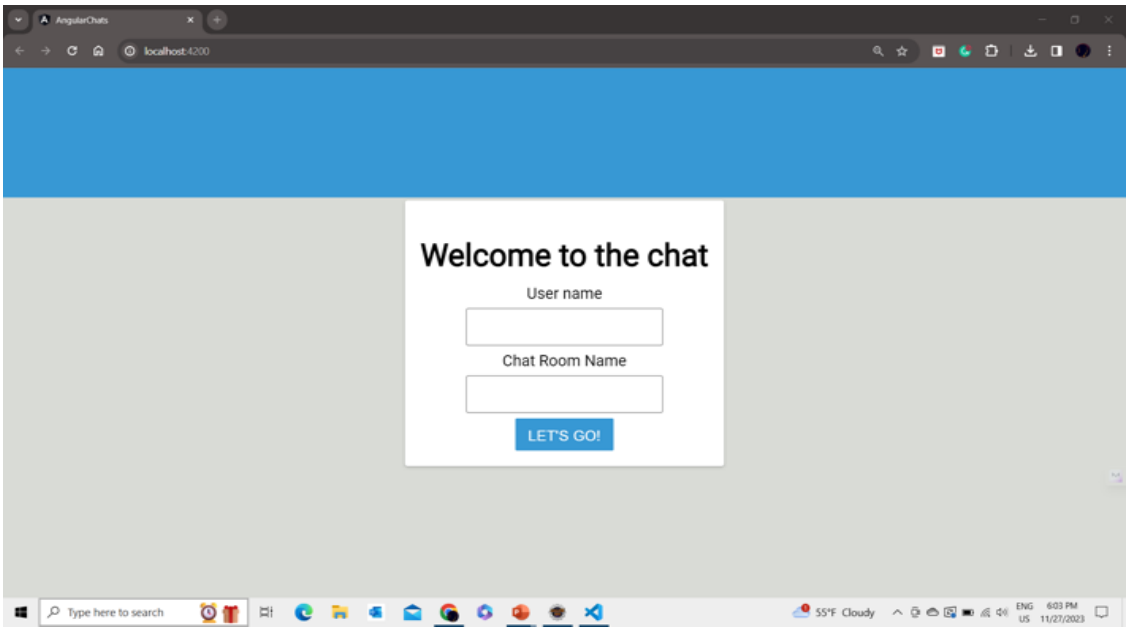
Availability is still another important problem. The application needs to ensure that, in the case of an error, users can keep on chatting and that messages are not lost. The replication factor, which allows data to be replicated over several nodes—one node serving as the leader and capable of replicating data to multiple nodes—is one of Kafka's features. The fault tolerance of the application is made possible by this feature.

The ability to scale is the next obstacle. The program needs to manage a lot of data and heavy traffic. It could get more challenging to route data from clients to servers and vice versa as the number of users increases. To overcome this difficulty, data can be distributed and stored among users using Kafka topics and partitions, which enables the application to scale horizontally as traffic volume rises. Instead of relying on a single point of failure, Kafka functions as a cluster of message brokers to distribute data appropriately.

V.SYSTEM FUNCTIONALITIES

1.User login:

- Initially when a new users want to make use of chat applications, he/she as to create an account to login.
- If user is an existing user, he/she can directly login into the account with properly credentials.

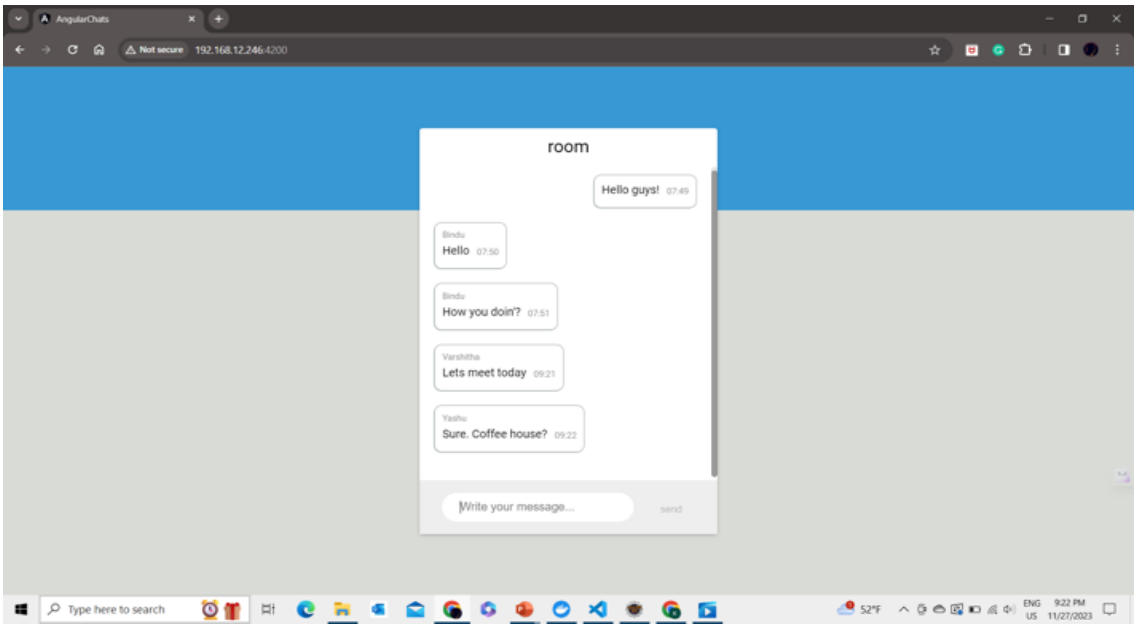


2.User authentication:

- To check if the user is a valid person or not, who is trying to access his/her account, we validate the user credentials before being logged into the chat rooms.
- Only the correct match of credentials a user can access the chat services.

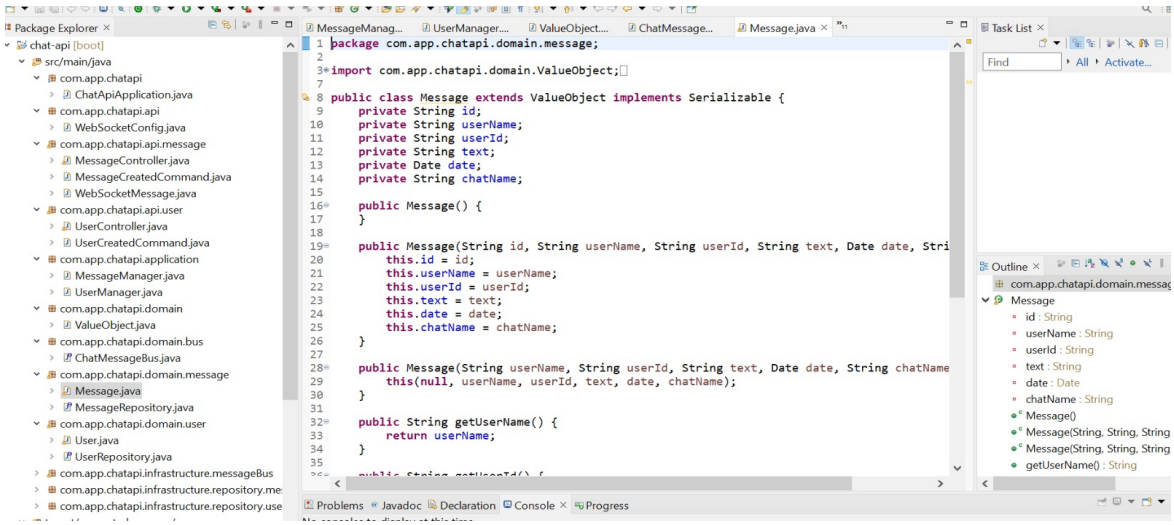
3.Real time messaging:

- We can start making use of chat services either sending message to either an individual or group of individuals in a group chat.
- There won't be any delay in message delivery.



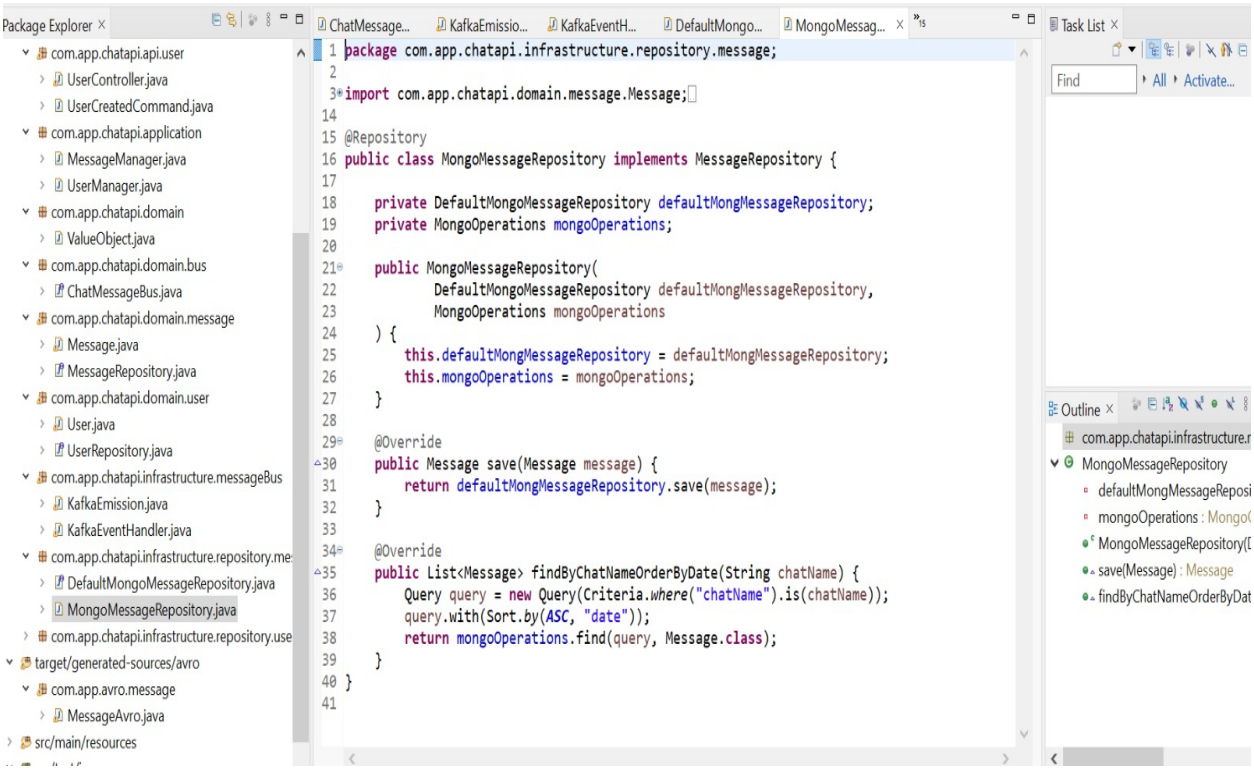
4.Message Queue:

- To prevent loss of messages in the network, we make use of Kafka's distributed feature of message queueing.
- So, the loss of messages is minimized, and lost message can also be retrieved.
- All the sent messages are delivered in the linear fashion of their generation timestamp.
-



5.Supports real-users:

- We have a dynamically growing database such as Mongodb as new users try to make use of the chatting service.
- The mongodb database is consistent and keeps up to date with user information.
-



VI. TECHNOLOGIES:

A. JavaScript programming language:

An interpreted, high-level programming language called JavaScript is used to create dynamic, interactive websites and web apps (Tomar & Dangi 2021, 1). One of the most popular programming languages on the web, it was first created in 1995 by Brendan Eich at Netscape in partnership with Sun Microsystems (McFarland 2008).

Before JavaScript was developed, web browsers were rudimentary programs that could display hypertext sites, according to Keith (2006, 1). Because of JavaScript's characteristics, web developers can add interactive elements to their pages, like pop-up windows, input validation, and keyboard and mouse click response. JavaScript is also used to create sophisticated mobile applications, browser games, and web applications.

Rich, interactive websites may be made with JavaScript, an adaptable language that works well with HTML and CSS. These days, ECMA is responsible for the development and maintenance of JavaScript, which is continuously changing and adding new features.

B. Angular framework:

A well-liked free and open-source JavaScript framework for creating dynamic user interfaces is called Angular (Fedosejev 2015, 2). A group of developers is currently responsible for maintaining it after Google developed it. AngularJS facilitates the creation of reusable user interface (UI) components and gives developers effective control over how the UI is updated in response to data changes or user interaction.

Angular's usage of virtual DOM (Document Object Model) speeds up the rendering and updating of user interface elements in web browsers, making it one of its key features. Additionally, the framework provides a declarative programming model, which saves developers time and facilitates the management of complicated UI components.

Angular is utilised by numerous tech titans and has been widely accepted in the web development world. The framework's versatility, ease of use, and interoperability with other well-liked front-end frameworks and libraries all contribute to its success. Angular is the second most popular technology, only behind NodeJS, according to a Stack Overflow (2022) survey of over 70,000 engineers.

C. Java programming language:

Sun Microsystems originally published Java, an object-oriented programming language based on classes, in 1995 (Savitch 2018, 49). Since then, it has grown to be one of the most extensively used programming languages worldwide, finding use in everything from embedded systems to desktop and web applications.

Java's security features, scalability, and support for distributed computing are what make it so popular. The "Write once, run anywhere" (WORA) tenet of Java design makes it platform-independent (Sufyan 2022, 2). Because Java code is compiled into bytecode, an intermediate form that may run on several platforms via the Java Virtual Machine (JVM), this capability is made possible. Additionally, Java is a perfect language for developing sophisticated applications because of its object-oriented capabilities. Java makes it simple for developers to organise and maintain big codebases by encapsulating data and behaviour objects.

Since Java is open-source, there is a sizable and vibrant developer community nowadays. Java development and maintenance have been sustained by Oracle Corporation since its 2009 acquisition of Sun Microsystem.

D. Spring Boot:

An open-source Java framework called Spring Boot makes the process of creating a web application easier and faster (Spring n.d.). Pivotal Software, the same organisation that created the well-known Spring Framework, produced it.

The fact that Spring Boot makes the setup and configuration required for a Java application simpler is one of its main advantages. Programmers can concentrate on developing code instead of configuring the application by using Spring Boot. Spring Boot's convention-over-configuration

methodology makes this feasible. Additionally, it offers developers a set of initial dependencies so they may quickly add features to their apps.

Web application development is meant to be flexible and efficient with Spring Boot. It provides a set of ready-to-use modules for developers to easily integrate into their projects. Among other things, these modules go over web services, web access, and security.

Because Spring Boot is compatible with microservices architecture, it is an all-around robust and flexible framework that can help developers create highly scalable web applications. The auto-configuration feature of Spring Boot allows developers to set up their services automatically based on the dependencies they utilise. This makes writing boilerplate code less necessary, which facilitates the development and upkeep of microservices.

E. MongoDB:

MongoDB is a NoSQL database which is also a cross-platform, document-oriented database system. MongoDB uses documents that resemble JSON. MongoDB offers various distributed features such as following:

Replication and fault tolerance:

MongoDB offers replication through managed service by default; if a cluster has several active hosts, it will automatically choose the primary replica to handle write requests. MongoDB will automatically choose a new primary replica from the available hosts when the primary replica is manually changed.

Fault tolerance:

The great majority of hosts in a cluster must be in good condition in order for them to be able to choose the primary replica automatically when necessary. Deploying clusters with an odd number of hosts is therefore more economical when utilizing Managed Service for MongoDB. A cluster consisting of three hosts, for instance, can function without a host. A cluster with four hosts can only lose one host at a time; if a second host goes down, there won't be enough hosts left to choose a new primary replica. For the same reason, a cluster with two hosts cannot

guarantee full fault tolerance since the lone surviving host is unable to designate itself as the primary replica.

Security Features:

1. MongoDB Authentication

Verification of the identity of an entity seeking to establish a connection is the process of authentication. MongoDB supports various authentication methods such as SCRAM, LDAP proxy authentication, Kerberos authentication. These protocols allow MongoDB to comply with various environments' requirements and interface with your current authentication system.

2. MongoDB Authorization

Finding out the precise permissions of the entity trying to connect is the process of authorization. Role-based access control is used by MongoDB to manage access. It makes it possible to assign one or more roles to each user, which controls who can access which database functions and resources.

3. MongoDB Auditing

MongoDB Enterprise has sophisticated auditing features. It records administrative actions (DDL), such as schema operations, authorization, read and write (DML) operations, and authentication, as well as access and actions made against the database. It helps in monitoring and maintaining a record of activities within the database.

Auditing increases the overhead of recording events, it may have an effect on database performance. Optimise audit settings to strike a compromise between security requirements and performance concerns.

When providing audit-related credentials, use caution because they may give rise to substantial access to private data. Always adhere to the least privilege concept by only allowing users or roles the minimal amount of access required to carry out their intended duties.

4. MongoDB Encryption

Administrators can encrypt data in backup repositories and permanent storage, both in transit and at rest, using MongoDB. While data is being utilised on the server, users have the option

to encrypt data at the field level, shielding private information from administrators and other authorised users.

5. Database Monitoring and Upgrading

Gaining the visibility required to guarantee performance, availability, and security in an IT infrastructure requires proactive monitoring of every component. Preventing potential defects from negatively affecting the system's performance is aided by it. Additionally, it assists in real-time exploit identification to lessen the impact of a security breach.

MongoDB comes with a number of tools that you may use to keep an eye on your database, such as mongostat and mongotop.

More than 100 database and system health measures, such as replication status, CPU and memory usage, operations counts, open connections, node status, and queues, may be monitored by Ops and Cloud Manager. In addition, when a host is accessible to the Internet, Cloud Manager notifies the user.

6. Create Separate Security Credentials

Create login credentials for every user or process that uses MongoDB in order to enable authentication. If more than one person requires administrative access to the database, do not share login credentials as this raises the possibility of account breach and complicates the task of monitoring administrative access. Give each person their own credentials and distribute permissions according to responsibilities.

7. Use Role-Based Access Control

Assign permissions to jobs like application server manager, database administrator, developer, and BI platform, rather than giving them to specific users. Predefined roles like clusterAdmin, dbAdmin, and dbOwner are provided by MongoDB. These positions can be further tailored to fit the requirements of particular groups and departments.

8. Encrypt Your Data

The data will be accessible to unauthorised users in the event of a data breach. Encrypting data renders sensitive information illegible to anyone lacking a decryption key, so mitigating the potential harm in the event of a data breach.

9. Use the Official MongoDB Packages

Installing MongoDB is simple because it is available as packages in the repositories of all common Linux systems. Make sure the package has passed stability tests and is a genuine MongoDB package, though. Furthermore, you must ensure that the community responsible for maintaining the operating system repository has the most recent MongoDB security patches installed.

For these reasons, MongoDB advises against utilizing operating system distribution-specific sources and instead suggests using their official package repositories.

10. Auditing and Logs

The who and when of modifications made to your database configuration are recorded in an audit trail. MongoDB Enterprise offers a comprehensive audit record of administrative actions through its auditing platform.

11. Apply MongoDB Security Fixes

Attackers search database systems continuously for new weaknesses. As a result, it's critical to stay informed about security updates and issue patches that MongoDB maintainers publish. Visit MongoDB's dedicated alerts website to receive real-time notifications regarding security updates and vulnerabilities.

F. Apache Kafka:

Originally created by LinkedIn in 2011, Apache Kafka is an open source distributed streaming technology (Garg 2013, 22). The original goal of LinkedIn's design was to create a fault-tolerant, scalable platform for processing and storing massive amounts of data in real-time.

The distributed commit log concept, which enables real-time data processing and storing across a cluster of workstations, is the foundation of Kafka (Thein 2014, 9478). As a result, millions of

events can be processed by Kafka every second. Several major corporations use Kafka globally, such as Cisco, Goldman Sachs, Box, and many more (Apache Kafka n.d.). Kafka processes seven trillion messages a day at LinkedIn (LinkedIn Engineering, n.d.).

Additionally very flexible, Apache Kafka can be applied to a wide range of use cases. Kafka can be used as a messaging system to transfer data between servers, as well as a streaming platform to analyse data in real time and a storage system to store enormous amounts of data.

The general architecture of Apache Kafka is shown in Figure 2. Furthermore, as seen in Figure 2, message publishers post to a topic (which might be managed by several brokers) and message recipients watch for messages from that topic. Kafka determines which brokers will conduct the read-and-write operation for each topic partition by putting the theory of leader election into practice. The next section will go into deeper detail on Apache ZooKeeper and leader election theory.

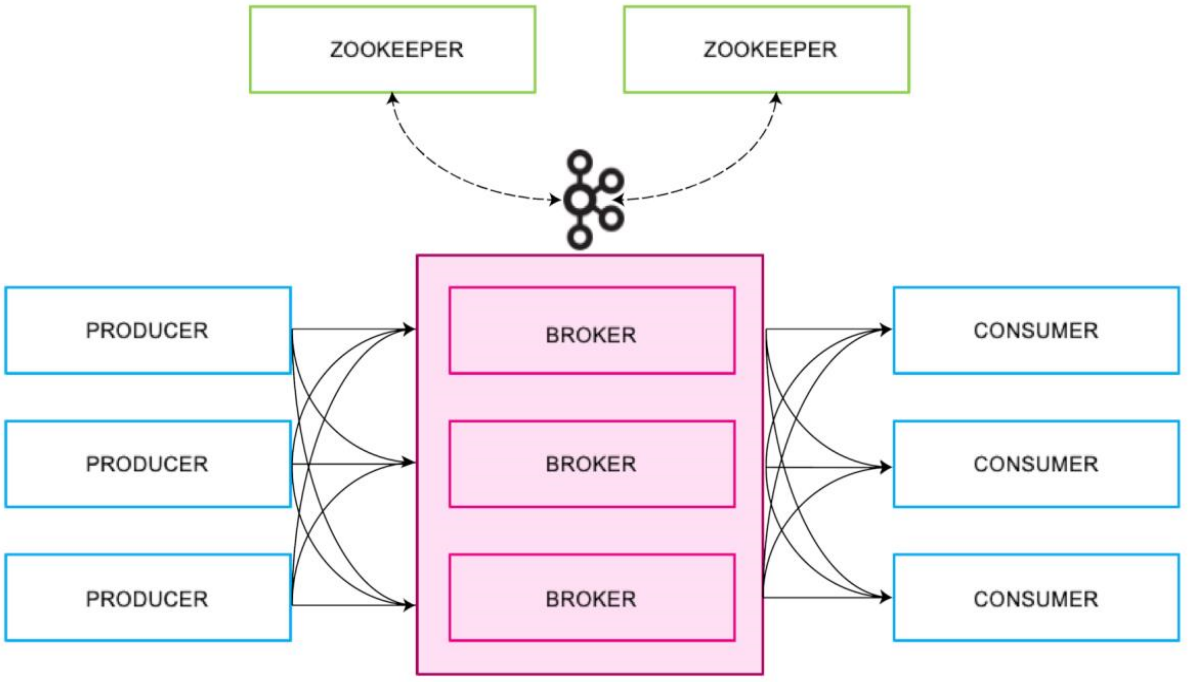


FIG 2. Kafka architecture with ZooKeeper.

G. Theory of leader election

One basic issue in distributed computing is leader election, where a set of nodes has to decide on a single leader node to guide the group's activities (Dolev, Israeli & Moran 1997, 424). In order to facilitate effective decision-making and coordination, the election of a leader serves to guarantee that there is a single point of control inside the system (EffatParvar et al. 2010, V2-6).

Because nodes can malfunction or go down at any time, causing network splits and inconsistent system states, leader election can present a difficult issue. Thus, it is necessary to create a scalable and fault-tolerant leader election theory.

To keep a consistent view of the cluster membership and identify the broker leading each partition, Kafka makes use of the ZooKeeper coordination service. ZooKeeper will identify the problem and start the leader election process when a broker goes offline. The remaining brokers in the cluster are able to communicate with one another and come to a consensus over who should take the lead going forward thanks to the ZooKeeper coordination service.

I. Apache Kafka Data Consistency and Fault Tolerance:

Although Kafka is best recognized as a streaming platform for processing data in real-time, it also has strong capabilities that make it an excellent choice for ensuring data consistency amongst dispersed systems. Let's investigate a few of these attributes:

1. Publish-Subscribe Model: Messages are sent to certain subjects by data producers (publishers), and subscribers (subscribers) subscribe to those topics to receive the messages. This architecture ensures constant data flow by facilitating smooth coordination and communication between various components.
2. Fault Tolerance and Replication: Fault-tolerant architecture is a feature of Kafka. In a Kafka cluster, it replicates data among several brokers (servers), guaranteeing high availability and durability. The system immediately switches to the replicated data in the event of a failure, protecting data and guaranteeing consistency.
3. Message Ordering and Retention: Kafka ensures that the message is processed in the order in which it is received by guaranteeing message ordering inside a partition. When several events or modifications take place at once, this capability is essential for preserving data

consistency. Furthermore, Kafka keeps messages for a configurable amount of time, letting users consume data at their own speed and guaranteeing steady data availability.

4. Exactly-Once Semantics: In order to guarantee that messages are processed just once, regardless of possible failures or retries, Kafka offers support for exactly-once semantics. This function guarantees dependable and consistent data processing by removing duplicate data problems.
5. Event-Driven Architecture: The event-driven design of Kafka encourages loose connectivity between its parts. Organisations can guarantee data consistency by using event-driven communication patterns, which allow for real-time synchronisation by spreading events and updates across many systems.

J. Kafka Security

Data from the Kafka can be read by several consumers, and data can be produced by multiple producers. But protecting data security is equally essential.

The necessity of Kafka security

There are a few factors that explain why security is necessary:

1. Multiple users may read the same piece of info. In certain situations, the user may choose to share data with one or two particular customers. As a result, the data must be protected from other users.
2. Customers are free to submit information or messages on any subject. As a result, any undesired customer could destroy the user's current customer base. It needs authorization security and is terrible.
3. Additionally, any topic within the user cluster could be removed by an unauthorized user.

Components of Kafka Security:

Mainly, there are three major components of Kafka Security:

1. Encryption**:** Encryption protects the data sent between customers and the Kafka broker. Kafka encryption makes assurance that data cannot be intercepted, stolen, or read by another client. The information will only be distributed in encrypted form.

2. Authentication**:** This guarantees that different apps can connect to the Kafka brokers. Only approved applications will have the ability to post and read messages. For the purpose of accessing the message from the cluster, the approved apps will have their unique user ID and password.

3. Authorization**:** This is the step that follows authentication. Upon successful authentication, the client can either publish or consume messages. To avoid data contamination, the programs might also be prohibited from write access.

K. Apache ZooKeeper:

It is a centralised infrastructure that handles configuration, synchronisation, and naming services in a distributed system offered by Apache ZooKeeper, a distributed coordination service (Apache ZooKeeper, n.d.). The Apache Software Foundation is in charge of maintaining this open-source initiative. Furthermore, ZooKeeper is built to be scalable and fault-tolerant.

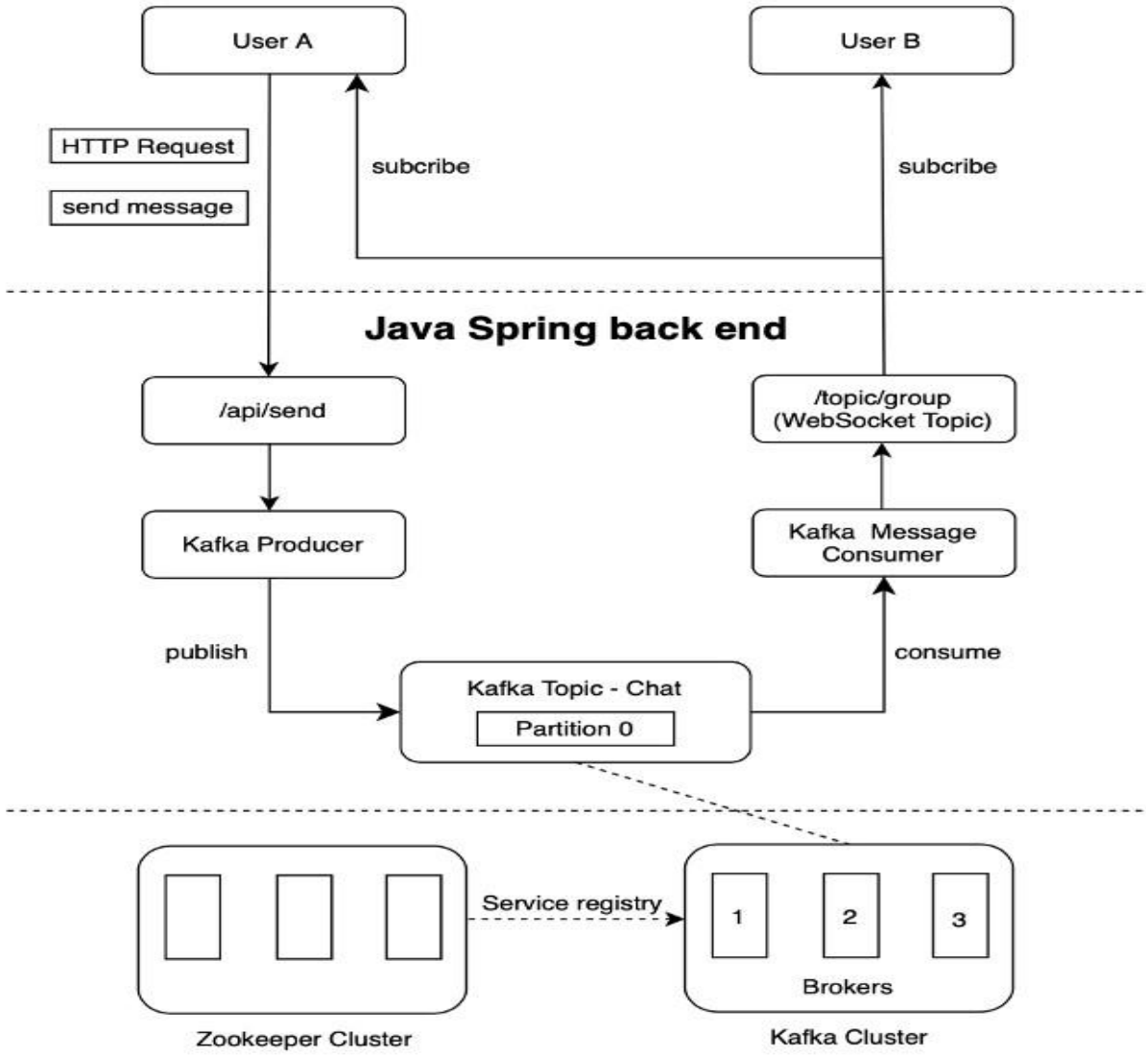
One of the most important aspects of ZooKeeper is its coordination service, which uses a consensus mechanism to manage the system's status even in the event of a failure. ZooKeeper serves as both a broker registry and a monitoring and failure-detection tool for Kafka. ZooKeeper notifies Kafka when a broker fails and Kafka reassigns data to the surviving brokers in the cluster.

VII. DESIGN AND IMPLEMENTATION DETAILS OF SYSTEM ARCHITECTURE

A. Software Architecture

Users can create an HTTP request to the API endpoint from the client-side of the chat application with their messages, usernames, and timestamp included. The pub/sub (publish/subscribe) paradigm in Kafka refers to a communications pattern in which producers, or publishers, send messages to a subject in the chat room, and consumers, or subscribers, receive messages from that topic. The overall architecture of the implemented program is depicted in Figure 3. When the server receives messages from users, it forwards them to the Kafka producer, who then posts them to the Kafka topic (Figure 3).

A Kafka topic is a feed name or category that Kafka producers post messages to, which Kafka consumers can read and utilise. The below figure demonstrates how the kafka consumer takes message that are sent by the kafka producer and how kafka topic stores and transfers the messages



.

FIGURE 3. Chat application architecture.

ZooKeeper coordination service registration is required for Kafka brokers, who oversee Kafka topics (Figure 3).There is only one leader broker per partition and it will handle the read and write operations and the other brokers serve as followers, replicating data in the event of a failure.

Additionally, by handling all HTTP requests submitted to API endpoints with predefined annotations, Spring Boot's event-driven design streamlines the logic of the application (Picture 1).

*MessageController.java

```java
package com.app.chatapi.api.message;

import com.app.chatapi.application.MessageManager;
import com.app.chatapi.domain.message.Message;
import jakarta.validation.Valid;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

// HTTP GET/POST requests for Chat App

@CrossOrigin(origins = {"http://localhost:4200", "http://localhost:5173", "http://localhost:3000"}, maxAge = 3600)
@RestController
@RequestMapping("/messages")
public class MessageController {
    private final MessageManager messageManager;

    @Autowired
    public MessageController(
            MessageManager messageManager) {
        this.messageManager = messageManager;
    }

    //Get the Chat Data From Kafka
    @GetMapping("/chat")
    public List<Message> getMessages(@RequestParam String chatName) throws Exception {
        return messageManager.getAllMessages(chatName);
    }

  //Post the Chat Data to Kafka
    @PostMapping("/new")
    public void addNewMessage(
            @Valid @RequestBody MessageCreatedCommand command
    ) {
        try {
            Message message = messageManager.save(command.getUserId(), command.getText(), command.getchatName());
            messageManager.send(message);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

}

```

PIC 1. GET/POST requests with Spring Boot.

B. Kafka configuration:

There is only one subject produced in the implemented program, which is the chat topic. A Kafka topic is a feed name or category that in Kafka represents a stream of data. It is a logical container that both Kafka producers and consumers utilise to post and receive messages. A subject can have more than one partition to allow for fault tolerance and parallelism. Each partition is recognized by its own name.

A Kafka partition is a portion of a topic that has an unchangeable, ordered message sequence within it. On the other hand, within a topic, there is no global ordering between Kafka partitions. To ensure fault tolerance and high availability in a Kafka cluster, each partition is duplicated over several brokers.

For Kafka consumers, the maximum parallelism of message consumption depends on the number of partitions in a topic. If a message key is not supplied, Kafka distributes messages among partitions using a partitioning technique based on a hash of the message key. Since the application only has one group chat, In the chat subject of the implemented application, the number of partitions was set to one. The number of partitions in the chat subject can be extended to accommodate many group chats. Each group chat can also have a message key, which allows the producer to send messages to the chosen partition.

C. Message model

The format of a message that can be sent between services is called a message model. It outlines the communication and organisation of data among various components. A message model also describes the content and structure of the messages that are being sent, including the data types, data elements, and any extra metadata.

The implemented application's message model, shown in Picture 2, contains the message's sender, content, and timestamp. The message model also specifies the payload that will be transmitted between the client and server.

*Message.java

```java
package com.app.chatapi.domain.message;

import com.app.chatapi.domain.ValueObject;

/* A message model specifies the structure and content of the messages being exchanged,
including the data element, data types, and any additional metadata.  */

public class Message extends ValueObject implements Serializable {
    private String id;
    private String userName;
    private String userId;
    private String text;
    private Date date;
    private String chatName;

    public Message() {
    }

    public Message(String id, String userName, String userId, String text, Date date, String chatName) {
        this.id = id;
        this.userName = userName;
        this.userId = userId;
        this.text = text;
        this.date = date;
        this.chatName = chatName;
    }

    public Message(String userName, String userId, String text, Date date, String chatName) {
        this(null, userName, userId, text, date, chatName);
    }

    public String getUserName() {
        return userName;
    }

    public String getUserId() {
        return userId;
    }

    public String getText() {
        return text;
    }

    public Date getDate() {
        return date;
    }

    public String getId() {
        return id;
```

PIC 2. Message model.

D. Producer and consumer implementation:

Producers in Kafka post messages to a Kafka topic without being aware if they will be received by any consumers at all. Subscribed customers will receive all published messages on one or more of the topics. The distributed system's various components can communicate in a flexible and scalable manner because to the separation of producers and consumers. Additionally, the application can conceal system logic from the client side thanks to this functionality.

Between producers and consumers, Kafka brokers serve as middlemen in the publish-subscribe business model. Producers transmit messages to brokers, who forward them to every subscribed customer. Three Kafka brokers have been created and the replication factor set to three in order to establish fault tolerance in the chat application. Here, the data will be duplicated to two other

brokers, and one broker will serve as a leader in charge of read and write activities. This guarantees that the messages are accessible and will not be lost in the event that the leader broker fails.

E. Producer implementation

The configuration of the Kafka producer with Spring Boot is shown in Picture 3. As illustrated in Picture 3, a variable named "configs" was made in order to configure a Kafka producer. This variable can be supplied to the Kafka producer as a container for key-value pairs. The bootstrap server addresses are the first key-value pair to be added. The producer will use this pair to create an initial connection to the whole Kafka cluster. It provides a list of Kafka addresses. Almost every Java object of any kind can be used as a value in Kafka, and any Java object of any kind can be used as a key. To send an object over a network, users must instruct the Kafka library on how to serialise each type of key and value into a binary format. As a result, in Picture 3, String serialise and Json serializer were selected for the key and value serializer classes, respectively.

Furthermore, a particular kind of Java bean called kafkaTemplate bean was developed to carry out tasks like sending messages to Kafka topics (Picture 3). Java Bean is a program element that is reusable, and that adheres to a set of standards for events, properties, and methods.

The handling of messages and their publication to the Kafka topic thereafter required an endpoint (Picture 1). The previously configured kafkaTemplate bean (Picture 3) is subsequently injected and transmits messages to Kafka topic whenever a user creates a Post request to api/send.

F. Consumer implementation

Similar to this, in order to establish producers, we must first create a consumers configuration that includes consumer groups, Kafka addresses, and a deserializer for deserializing data from Kafka topics. Figure 3 shows the configuration of two beans: kafkaListenerContainerFactory and consumerFactory.

```yaml
kafka:
  bootstrap-servers: "http://localhost:9092"
  producer:
    key-serializer: org.apache.kafka.common.serialization.StringSerializer
    value-serializer: io.confluent.kafka.serializers.KafkaAvroSerializer
  consumer:
    group-id: "chat-api"
    auto-offset-reset: earliest
    key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
    value-deserializer: io.confluent.kafka.serializers.KafkaAvroDeserializer
    enable-auto-commit: true
  properties:
    specific.avro.reader: true
    schema.registry.url: "http://localhost:8085"
```

PICTURE 4. Producer and Consumer configuration.

As a result, messages from a topic can now be consumed using the @KafkaListener annotation. The figure below showcases the snippet to convert and broadcast messages to the WebSocket topic after ingesting messages.

KafkaEventHandler.java

```java
package com.app.chatapi.infrastructure.messageBus;

import com.app.chatapi.domain.message.Message;

@Component
public class KafkaEventHandler {

    private SimpMessagingTemplate template;

    @Autowired
    public KafkaEventHandler(SimpMessagingTemplate template) {
        this.template = template;
    }

    /* @KafkaListener annotation can now be used to consume messages from a topic */
    @KafkaListener(topics = "chat")
    public void listen(MessageAvro messageAvro) {
        Message message = new Message(messageAvro.getId(), messageAvro.getUserName(),
                messageAvro.getUserId(), messageAvro.getText(), new Date(Long.valueOf(messageAvro.getDate())), messageAvro.ge
        template.convertAndSend("/chat/"+messageAvro.getChatName(), message);
    }

}
```

PIC 5. Consumer messages.

G. Testing

JMeter was used to test the application. The technique of mimicking user traffic to an application in order to assess how well it functions under various loads is known as load testing. With JMeter, one may design test plans that replicate a large number of customers submitting queries to an application in order to gauge throughput and response time.

H. Meter configuration

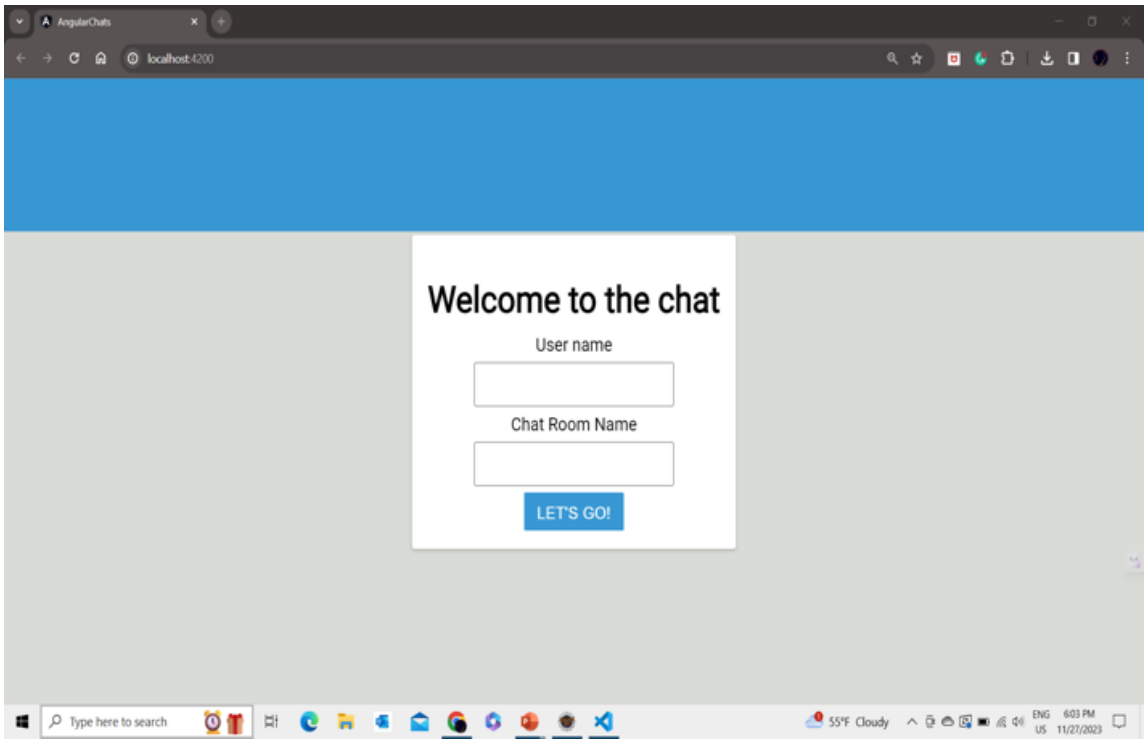
The message content and username were included in the HTTP requests. 10,000 requests would be processed in 100 seconds because the ramp-up period was set to 100 seconds.

I. Basic Function

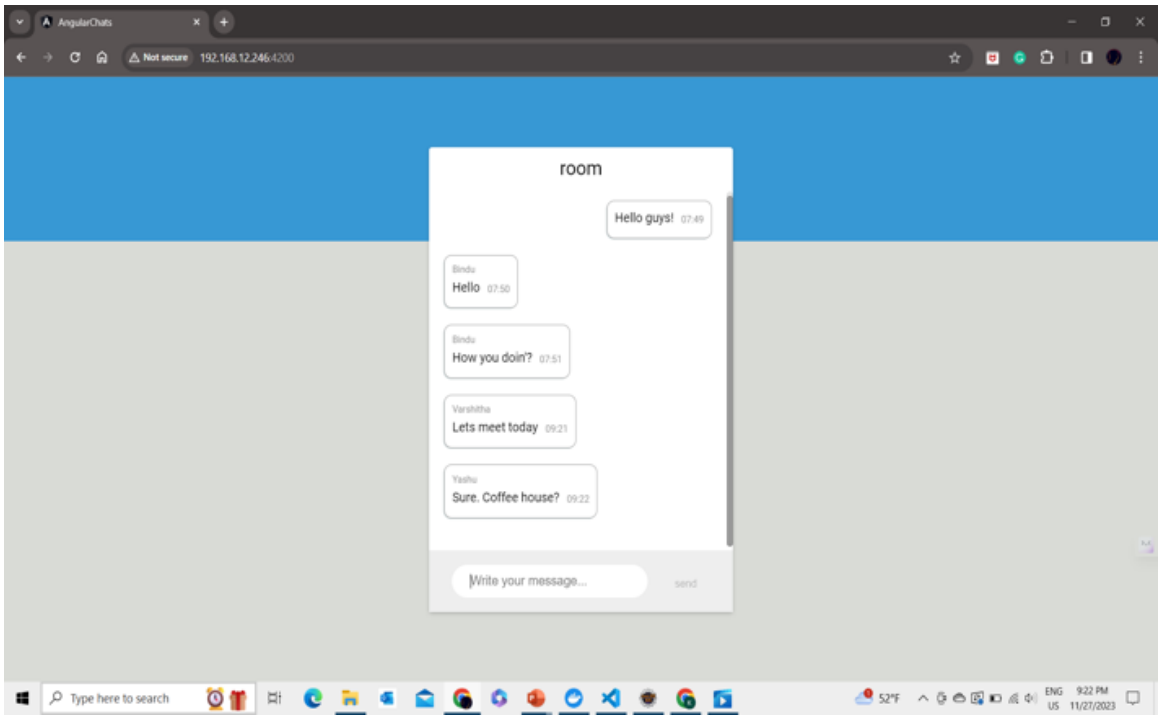
Real-time communication is possible with chat-based systems. They also give the encounter a more intimate feel. developing a chat application that enables several users to sign in on different browsers and communicate with one another on various channels.

Instantaneous delivery of messages is expected. It's a powerful messaging platform that facilitates user communication in real time. When utilizing the HTTP protocol, data delivery will be delayed since HTTP must complete a series of steps, including a client request and a server response. WebSocket protocol can be utilized to enable real-time communication between users, which would solve this issue. When using WebSocket, the client sends a request to the server and waits for a response before establishing a connection with the server. Users can get messages quickly from the chat application and polling can be avoided by using the WebSocket protocol.

Screenshots of final application:



PICTURE 6. Home Page



PICTURE 7. Chat Room

VIII. DESIGN AND IMPLEMENTATION DETAILS OF DISTRIBUTED FEATURES

The distributed features that Kafka, MongoDB, and Spring Boot technologies support are as follows:

1) Distributed Features exhibited by Kafka (Apache Kafka):
   - Distributed Messaging: Designed to handle real-time data flows, Kafka is a distributed streaming infrastructure. Distributed publish-subscribe messaging is supported.
   - Scalability: Kafka can accommodate growing loads and storage needs by adding additional brokers because it is horizontally scalable.
   - Replication: To guarantee the fault tolerance and high availability, data is replicated among several brokers. Resilience can be specified for this replication factor.
   - Partitioning: Kafka divides its topics into partitions to facilitate parallelism and scalability. It is possible to spread each split among several nodes.

2) Distributed Features exhibited by MongoDB:
   - Sharding: Sharding is a technique used by MongoDB to divide data among several machines. By spreading data among clusters, it allows for scalability and performance through the horizontal partitioning of data into shards.
   - Replication: MongoDB provides replica sets, which ensure high availability and fault tolerance by replicating data across many servers.
   - Horizontal Scalability: Scaling out is possible by adding more nodes to a cluster and distributing data and workload among these nodes.
   - Automatic Fail-over: In MongoDB, replica sets offer automated fail-over, which makes sure that one of the secondary nodes takes over as the primary in the event of a primary node failure.

3) Spring Boot:
   - Microservices Support: Spring Boot makes it easier to build microservices-based applications, enabling for distributed application development.
   - Service Discovery: It combines with service discovery systems such as HashiCorp Consul or Netflix Eureka to allow micro-services to locate and interact with one another in a distributed setting.
   - Distributed Configuration: For distributed systems, Spring Cloud Config offers centralised external configuration management that enables configuration changes without requiring service re-deployments.
   - Load Balancing: Load balancing components are part of Spring Cloud, allowing requests to be split among several instances of a service.

With the help of these technologies, developers may create scalable, fault-tolerant, and highly performant distributed applications in a variety of distributed environments. Within a distributed system, they address several elements such as messaging, scalability, fault tolerance, data dissemination, and service communication.

Distributed features implemented in our chatting application:

A) Process Communication and Process Coordination through serialization
   - All the generated message are communicated between the users without being the loss of data.
   - Messages are delivered in sequential fashion of their generation.
   - Messages are delivered in real time without any time lag.
   - Process are well coordinated for seamless delivery of the messages.
   - Support single person chat rooms or group chat rooms

B)Fault Tolerance
   - Kafka and Mongodb are fault tolerant in nature.
   - They spin up new nodes/clusters in case of failure in the current node.
   - A backup node is maintained(as replica) to support in time of failures and disaster recovery.

- Kafka's fault tolerance is essential to safeguard the generated messages from being lost.
- Mongodb's feature helps in loss of users data with all the user credentials.
- Usually standby node or replica node replaces the failure node until the current node reinstates back to functioning.

C)Scalability

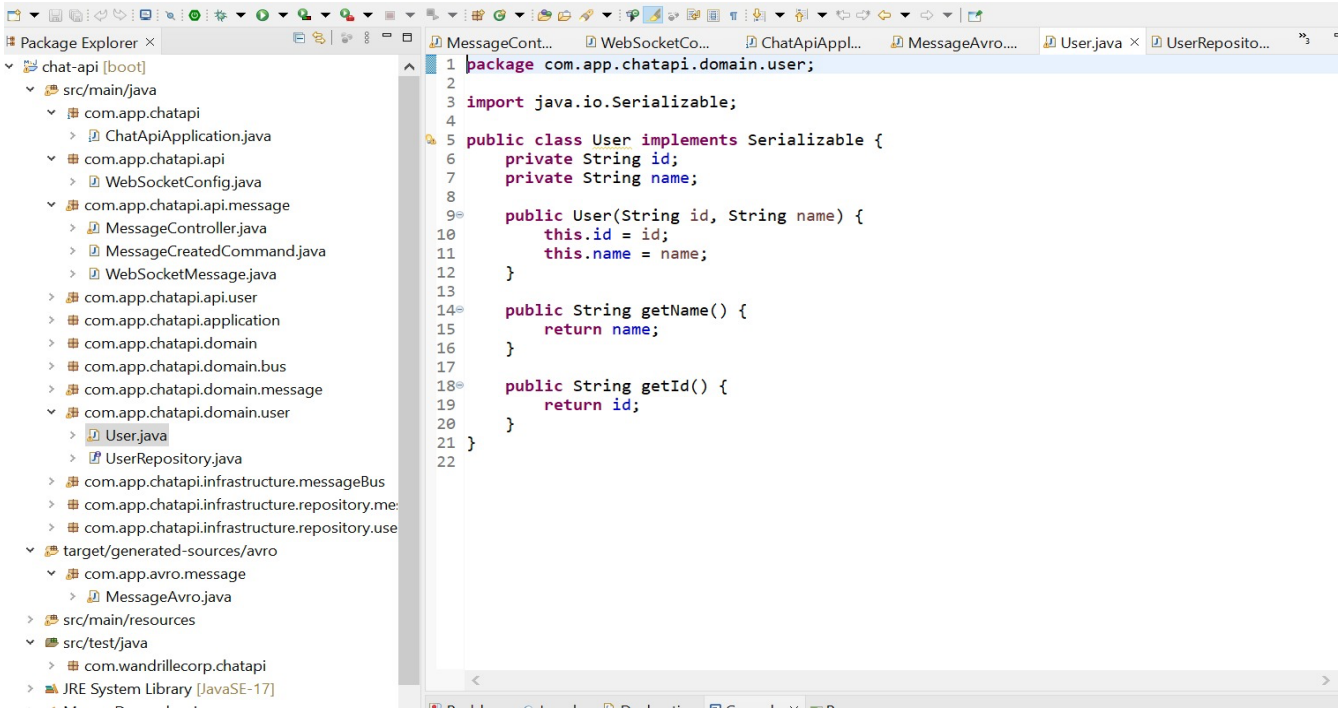
- Kafka and mongodb supports scalability.
- Kafka scalability is to handle large volume of messages deliveries over the network.
- Mongodb's scalability is to handle to dynamically growing users that utilize the chatting service.
- A replica node of current functioning node gets activated on the increasing volume of users or messages.
- Scaling supports increasing and decreasing of messages or users.

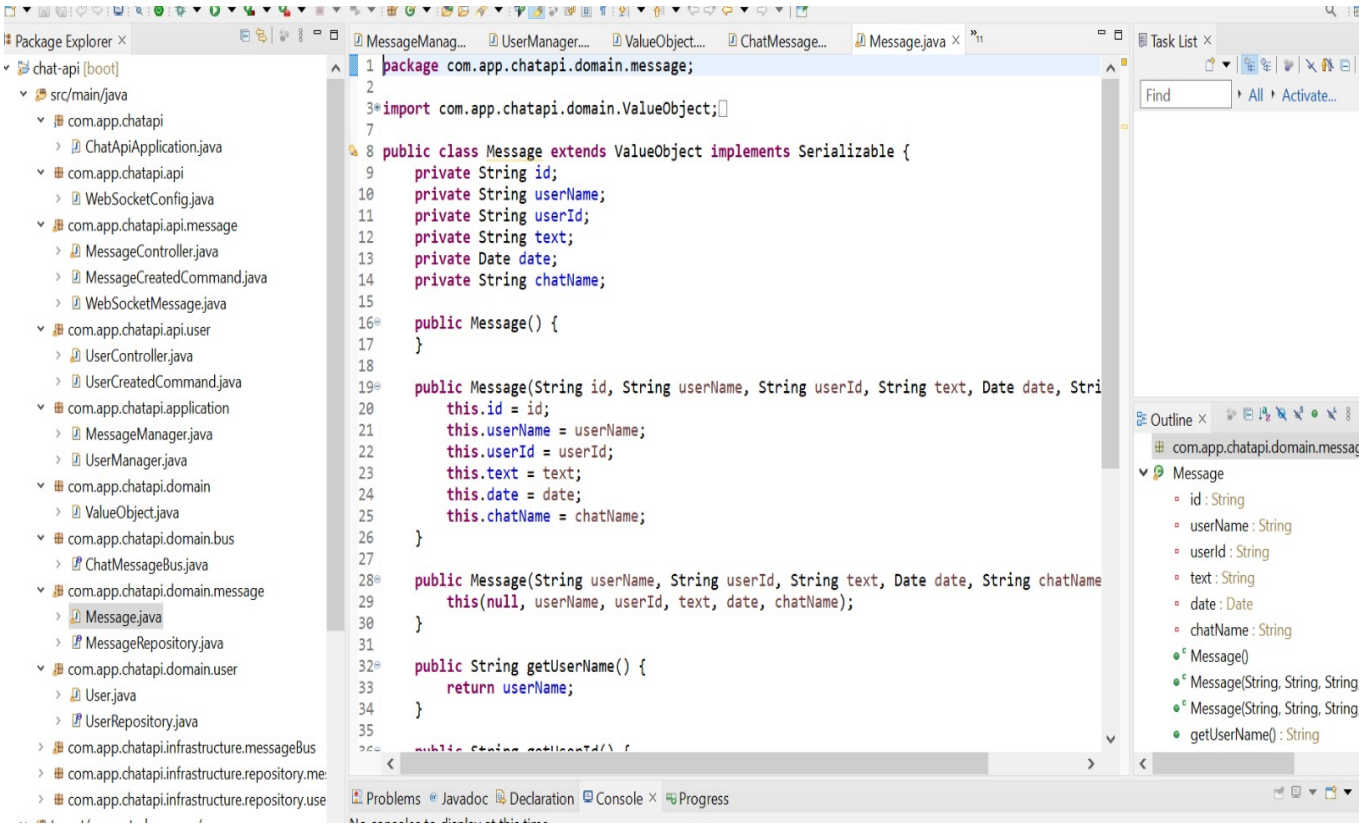
D)Consistency

- Mongodb supports consistency to maintain users data.
- Users tend to change their credentials when they might feel potential threat to their security, in order to keep up with changes in credentials and helps users to login in back to their chat rooms successfully, we have to supports new changes and maintain data consistently.
- So mongodb always keeps with updating data and providing seamless access to chat rooms.
- Kafka also supports consistency but we are not leveraging the service in the chat application.

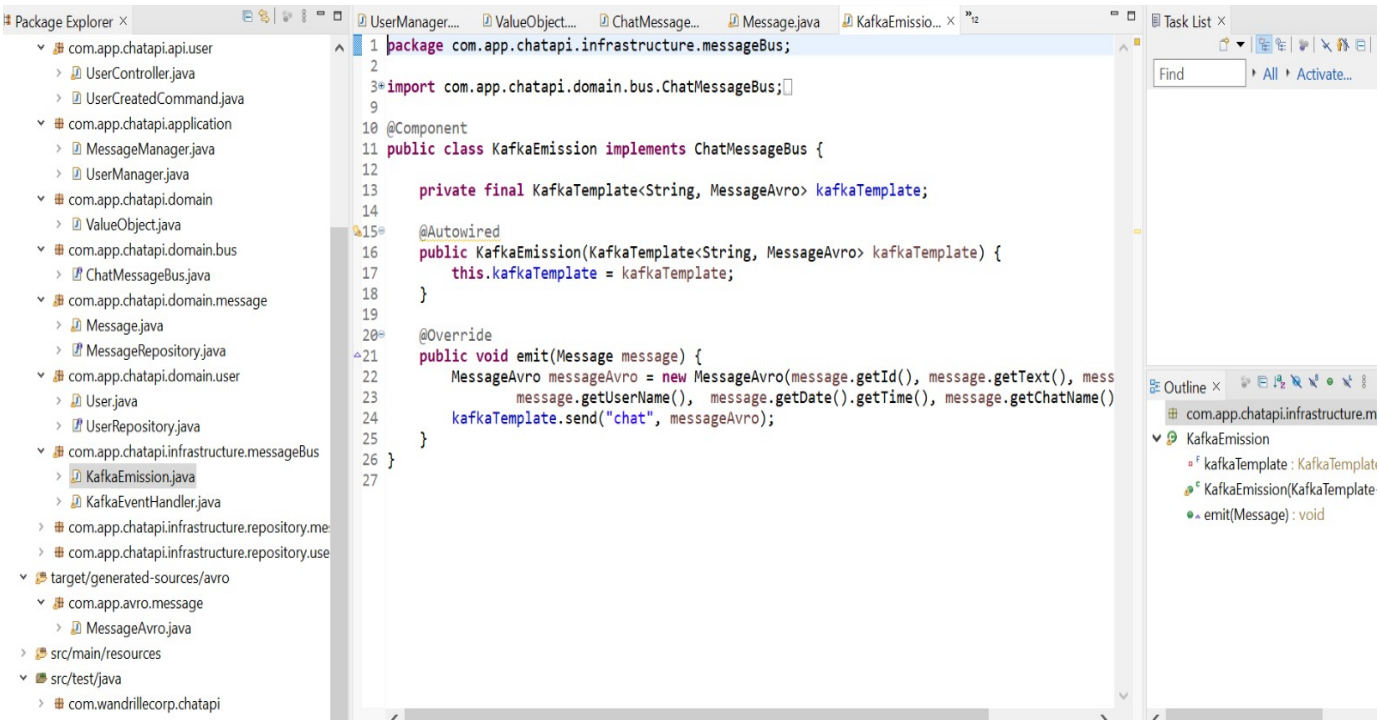
The following are the code snippets demonstrating the distributed features such as serialisation, consistency, replication and scalability, and fault tolerance.

Serialising for user and message:

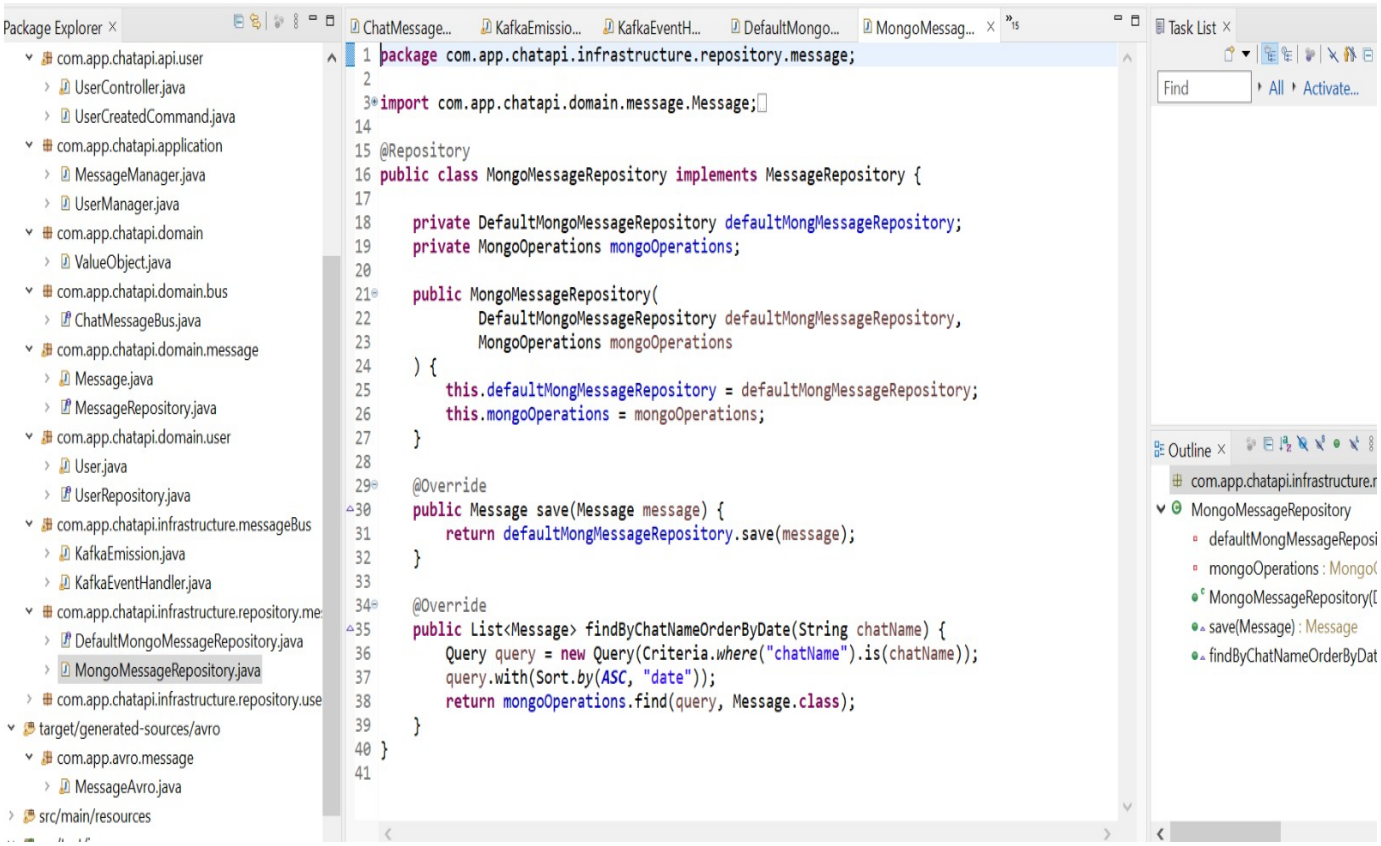


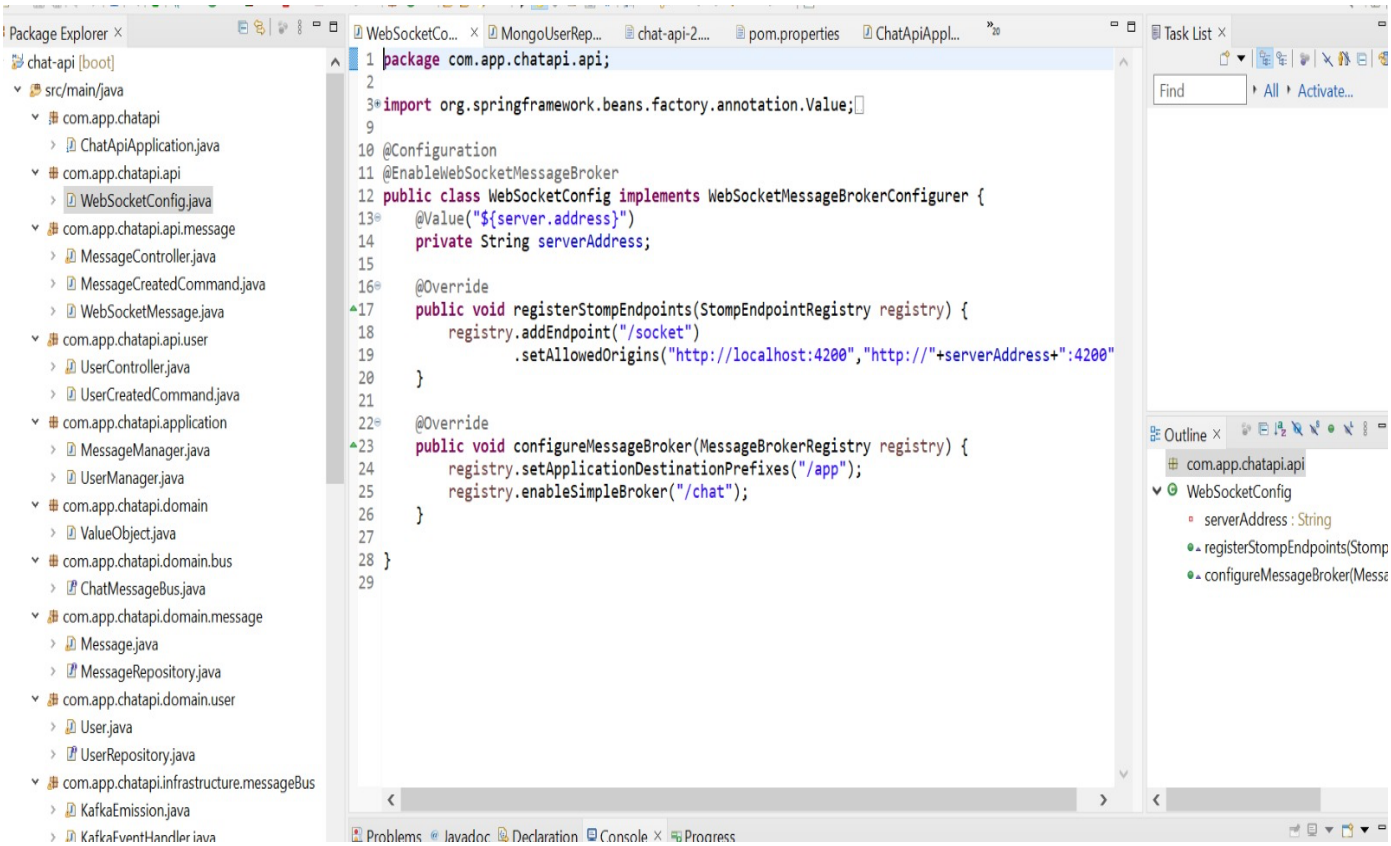


Kafka: (Provides Built in Replication, Communication, and Fault Tolerance)



MongoDB: (Provides Consistency, Reliability,scalability)





IX. REFLECTION

We learnt how to integrate Kafka, Spring boot, Angular, MongoDB, and other technologies using Docker.

We understood the concept of decentralization and distributed while exploring the technologies how are different from traditional systems and how they benefit while dealing huge volume of data.

Kafka is a better option to implement for chatting application, and which indeed implements almost every distributed feature. It was challenging to set up and learn Apache Kafka. It was not easy to set up and manage Apache Kafka. We must determine the needs for networking, configure the appropriate interfaces, and implement security segregation. After everything is operational, we had to handle Day 2 operations and be able to identify and fix issues as they come up. It's not very

developer-friendly to use Apache Kafka. It could be challenging for people who are unfamiliar with Apache Kafka to understand the concepts of topics, logs, partitions, brokers, and so forth.

There is a significant learning curve. Learning the fundamentals of Kafka and the essential components of an event streaming architecture will require a great deal of training. To comprehend this specific challenge more fully, we must first become acquainted with Kafka's methods.

Overall, Apache Kafka is a system that streams data from sources, also known as producers, to targets, also known as consumers. Our data is published and stored in what we refer to as Kafka topics, which producers can push data into, and consumers can retrieve data out of.

A Kafka topic's contents consist of key-value pairs, each of which can have its value serialised into one of several data formats, including Avro, JSON, or Protobuf. What we refer to as a schema is the data format's structure. When attempting to encode data using different schemas, one issue that may arise is that consumers may not be able to comprehend producers; if the producer schema is different, your downstream consumers may begin to malfunction. Put otherwise, Kafka lacks data verification. Protocol support is another area where developers may encounter difficulties while integrating Kafka. Since Kafka operates within the Java Virtual Machine (JVM) ecosystem, Java serves as the client's primary programming language. If, for example, Python or C are your favourite languages, this can be an issue. Although there is multilingual open-source clients available, Kafka is not included with them. For complete protocol support, you will need to manually install and update these drivers.

Initially when we integrated the kafka, mongodb and other essential technologies, here we are manually spinning up each and every technology, and it is time consuming and sometime we may as well forgot to spin up something, then realized the running a script to spin up all the necessary bootstrap functions once without encountering each and every thing every single time. So understood the importance of scripts and learnt dockers fundamentals to implement for our use-case.

First, we created single chat rooms but later had to create multiple chat rooms and multiple channels on professors suggestion which was quite a challenging task and finally implemented after understanding the creation of group chat rooms for some reference materials.

X. CONCLUSIONS

We have created a chatting application for the real time communication purpose for real time messaging using Apache Kafka. Kafka was used in the design and implementation of the suggested chat program. It is important to note that not all real-world conditions, such as message size and number of group conversations, could be covered by the tests. Because processing each message would take longer due to these variables, the suggested application's performance would suffer. All things considered, the outcomes demonstrate that Kafka is a dependable and an effective messaging platform for creating real-time chatting applications. The load testing result demonstrated that Kafka could manage high throughput and big message volumes, and the architecture and capabilities of Kafka are appropriate for this use case.

In conclusion, the creation of a chat application that makes use of Kafka has shown to be successful in creating a dependable and scalable messaging system. The chat program offered a way to manage heavy user traffic even during emergencies; by utilising the coordination service Zookeeper and Kafka topics, the system can bounce back from errors fast. Appropriate configuration and optimization can boost the suggested application's performance. The development of additional features like file sharing, numerous chat channels, and message encryption can be the subject of future research. Apache Kafka has laid a solid foundation for the chat application's future improvements. Finally, we have created a chart system where multiple people can communicate with each other using single or multiple channels and usage of Kafka made it a distributed system as it enabled effective communication, scalable, fault tolerant application.

XI. REFERENCES

1. https://www.geeksforgeeks.org/apache-kafka/
2. https://www.geeksforgeeks.org/spring-boot/
3. Kafka, Apache. n.d. Distributed by [website]. Peruse as of March 15, 2023. Powered by https://kafka.apache.org
4. Use Cases for Apache Kafka, n.d. [website]. Peruse as of March 15, 2023.
5. Uses #uses_messaging at https://kafka.apache.org/uses
6. Zookeeper Apache. n.d. Here at Apache ZooKeeper, welcome. On March 18, 2023, read.
7. Zookeeper at Apache.org
8. Barot, T., and Oren, E. (2015), Chat Apps: A Guide. The Tow Center for Online News. University of Columbia. Reports.
9. In 1997, Dolev, S., Israeli, I., and Moran, S. uniform dynamic election of a self-stabilising leader. 8(4), 424440, IEEE Transactions on Parallel and Distributed Systems.
10. Yazdani, N., EffatParvar, M., Dadlani, A., & Khonsari, A. (2010) EffatParvar, M.R. enhanced leader election methods in decentralised networks. 2010, V2-6–V2-10; 2nd International Conference on Computer Engineering and Technology.
11. In 2015, Fedosejev, A. React.js fundamentals. Limited by Packt Publishing.
12. Garg, N, Apache Kafka, 1st edition. Birmingham: Packt Publishing
13. Distributed Systems Edition 3 by Maarteen Van Steen
14. Innoinstant. 2022: Chat Applications: A Look at Their History, Present, and Future. [Weblog]. dated February 20, 2022. On March 22, 2023, read. The evolution of instant messaging apps can be found at https://blog.innoinstant.com/
15. M. Iqbal, 2023-03. Revenue and Usage Data for WhatsApp. issued on February 8, 2023.
16. Keith, J. (2006). Dom Scripting: JavaScript with the Document Object Model for Web Design. Apress, Berkely, Calif.
17. LinkedIn's Kafka Ecosystem, n.d. Peruse as of March 15, 2023. Streams: Kafka at https://engineering.linkedin.com/teams/data/data-infrastructure
18. In 2017, Manish, K., and Singh, C. Creating enterprise message queues through design and deployment for data streaming applications using Apache Kafka. first printing. Packt, Birmingham.

19. D. S. McFarland (2008), "JavaScript," First Edition. Beijing: O'Reilly/Pogue Press.
20. https://spring.io/why-spring
21. Stack Overflow. https://survey.stackoverflow.co/2022/#overview
22. https://www.koombea.com/blog/chat-appsevolving/