Sri Eshwar College of Engineering
Coimbatore | Tamilnadu
An Autonomous Institution
Affiliated to Anna University, Chennai

U23CB593 SYSTEM DESIGN LABORATORY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SRI ESHWAR COLLEGE OF ENGINEERING
KINATHUKADAVU COIMBATORE - 641202

Sri Eshwar College of Engineering
Coimbatore | Tamilnadu
An Autonomous Institution
Affiliated to Anna University, Chennai

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

BONAFIDE CERTIFICATE

Certified that this is the bonafide record of work done by

Name: Mr. /Ms. ....……

Register No: ……... of 3rd Year

B.E/B.Tech ……... in the

U23CB593 SYSTEM DESIGN LABORATORY during the 5th **Semester** of the academic year **2025 – 2026 (Odd Semester).**

Signature of faculty In-charge

Head of the department

Submitted for the practical examinations of Anna University, held on …….

Internal Examiner

External Examiner

Table of Contents

Average:
Average (in words)

Signature of the Faculty

Ex No : 1	Layered architecture diagram for a note-taking application using component diagrams
Date:	

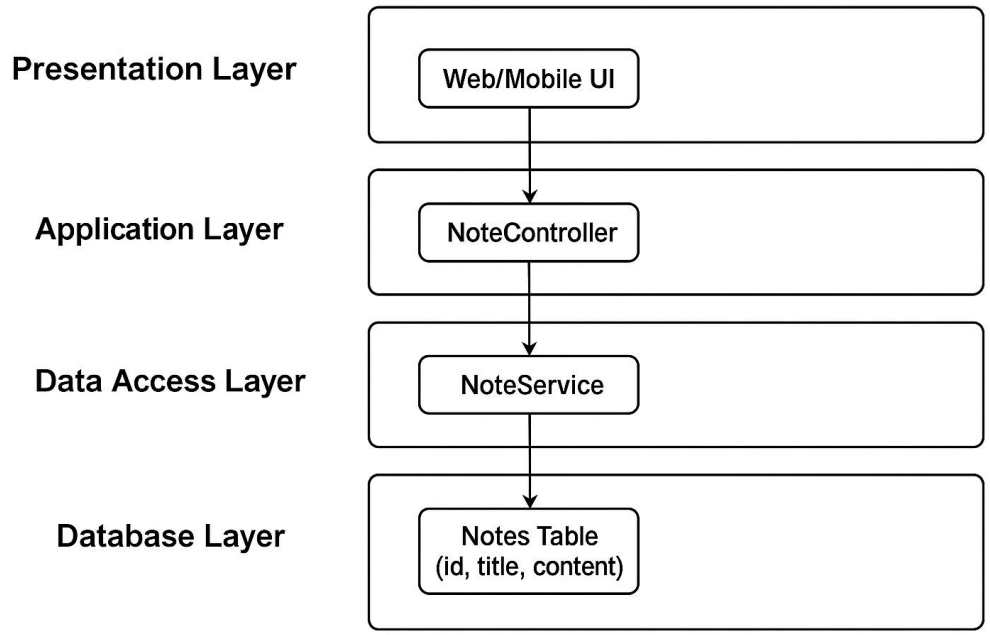
Aim:

To draw a layered architecture diagram for a note-taking application using component diagrams and explain the data flow between client, server, and database.

Procedure:

1. Determine the main components — client (UI), server (application logic), and database (data storage).
2. Assign specific roles to each layer — Presentation, Application, Data Access, and Database.
3. Represent each layer with components and connect them using arrows to show interactions.
4. Describe how requests move from client to server and responses flow back after database operations.
5. Ensure that all layers communicate properly and that the design supports scalability and maintenance.

Diagram:



Explanation :

1. Client to Server:

- The user interacts with the Presentation Layer (UI).
- For example, when creating a new note, the user enters text and clicks “Save”.
- The request (with note details) is sent to the NoteController in the server.

2. Server (Business Logic):

- The NoteController forwards the request to the NoteService, which contains the business rules.
- The NoteService validates the input and prepares it for database storage.

3. Server to Database:

- The NoteService calls the NoteDAO (Data Access Object), which executes SQL queries or ORM commands.
- The NoteDAO interacts with the Database Layer to insert, update, retrieve, or delete data.

4. Database to Server:

- The Database Layer returns query results to the NoteDAO, which passes them to the NoteService, and then to the NoteController.

5. Server to Client:

- The NoteController sends the response back to the UI.
- The UI updates the display — showing confirmation (for create/delete) or updated notes (for read/edit).

Result :

Thus the layered architecture diagram for the Note-Taking Application was successfully designed and the data flow between the client, server, and database was explained.

Ex No : 2	The High Level Design (HLD) and Low Level Design (LLD) for a URL Shortening service (TinyURL
Date:	

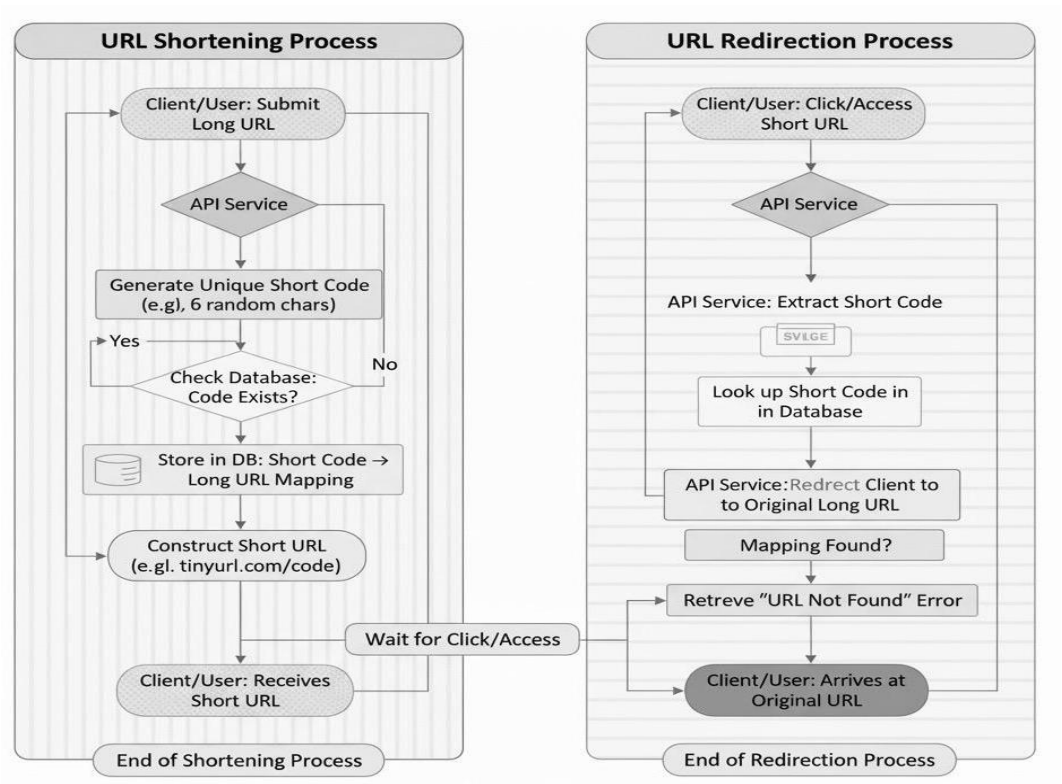
Aim:

To Prepare High Level Design (HLD) and Low Level Design (LLD) for a URL Shortening service (TinyURL), including major components, database schema, and flow diagrams.

Procedure:

1. Identify system components: API, Database, and Redirect Service.
2. Define data flow for URL creation and redirection.
3. Design database schema to store short and long URLs.
4. Implement short code generation logic using Base62 encoding.
5. Test URL creation and redirection with sample inputs.

Flow diagram:



Database Schema:

Field Name	Type	Description
id	INT (PK)	Unique identifier
long_url	VARCHAR	Original URL
short_code	VARCHAR	Generated short code
created_at	TIMESTAMP	Creation time

Program:

```
import java.util.*;

import java.util.concurrent.ConcurrentHashMap;


public class URLShortener {

    private static final String BASE_URL = "https://tinyurl.com/";

    private static final String CHARSET =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";

    private static final Map<String, String> urlMap = new ConcurrentHashMap<>();


    public static String shortenURL(String longUrl) {

        String shortCode = generateCode();

        urlMap.put(shortCode, longUrl);

        return BASE_URL + shortCode;

    }


    public static String redirect(String shortCode) {

        return urlMap.getOrDefault(shortCode, "URL not found");
```

```
    }

    private static String generateCode() {
        Random rand = new Random();
        StringBuilder code = new StringBuilder();
        for (int i = 0; i < 6; i++)
            code.append(CHARSET.charAt(rand.nextInt(CHARSET.length())));
        return code.toString();
    }

    public static void main(String[] args) {
        String longUrl = "https://example.com/some/long/path";
        String shortUrl = shortenURL(longUrl);
        System.out.println("Shortened: " + shortUrl);
        System.out.println("Redirected to: " +
redirect(shortUrl.substring(BASE_URL.length())));
    }
}
```

Output :

Shortened: https://tinyurl.com/aB9kLp

Redirected to: https://example.com/some/long/path

Result :

Thus the High Level Design (HLD) and Low Level Design (LLD) for a URL Shortening service (TinyURL), including major components, database schema, and flow diagrams was successfully designed.

Ex No : 3	The working of Redis cache for an API by explaining in-memory caching with TTL and LRU eviction policy
Date:	

Aim:

To illustrate the working of Redis cache for an API by explaining in-memory caching with TTL (Time-To-Live) and LRU (Least Recently Used) eviction policy.

Procedure:

1. Install Redis and connect it to your backend API (e.g., Node.js, Python Flask, or Django).
2. The API endpoint fetches data (e.g., product details) from a database.
3. Integrate Redis Cache:
    - Before querying the database, the API checks if the data exists in Redis.
    - If present, the cached data is returned (cache hit).
    - If not, data is fetched from the database and stored in Redis with a TTL.
4. Configure Redis to use maxmemory-policy allkeys-lru so least recently used keys are deleted when memory is full.
5. After TTL expires, the cache is cleared automatically, forcing the next request to fetch from DB again.

Program:

```
import redis.clients.jedis.Jedis;

import redis.clients.jedis.params.SetParams;

import java.time.Instant;

import java.util.Random;

import java.util.concurrent.TimeUnit;
```

```
public class RedisCacheDemo {
   // Simulated DB (slow)
   static class SlowDB {
      String fetch(String key) {
         try { TimeUnit.MILLISECONDS.sleep(200); } catch (InterruptedException ignored) {}
         return "VALUE_FOR_" + key + "_at_" + Instant.now();
      }
   }
   static class RedisCache {
      private final Jedis jedis;
      private final int ttlSeconds;
      RedisCache(Jedis jedis, int ttlSeconds) {
         this.jedis = jedis;
         this.ttlSeconds = ttlSeconds;
      }
      // Get value: Redis GET; fallback to DB fetch and SET with TTL
      String getOrLoad(String key, SlowDB db) {
         String val = jedis.get(key);
         if (val != null) {
            System.out.println("[CACHE HIT] " + key);
            return val;
         } else {
            System.out.println("[CACHE MISS] " + key + " -> loading from DB");
            String fromDb = db.fetch(key);
```

```
            // SET key with EX TTL atomically
            SetParams params = new SetParams().ex(ttlSeconds);
            jedis.set(key, fromDb, params);
            return fromDb;
        }
    }
    // helper to display approximate memory usage and key info
    void printInfo() {
        System.out.println("Redis INFO: " + jedis.info("memory").split("\n")[0]);
        System.out.println("Current TTL of sample key (if exists): " +
jedis.ttl("sample_key"));
    }
}
public static void main(String[] args) {
    // Adjust host/port if needed
    try (Jedis jedis = new Jedis("localhost", 6379)) {
        System.out.println("Connected to Redis: " + jedis.ping());
        // NOTE: Configure Redis server separately for LRU eviction:
        // CONFIG SET maxmemory 20mb
        // CONFIG SET maxmemory-policy allkeys-lru
        RedisCache cache = new RedisCache(jedis, 5); // TTL = 5 seconds
        SlowDB db = new SlowDB();
        // Warm some keys
        System.out.println("\n--- Warm cache ---");
        cache.getOrLoad("user:1001", db);
```

```
            cache.getOrLoad("user:1002", db);
            // Access pattern: simulate hits & misses
            System.out.println("\n--- Access pattern ---");
            cache.getOrLoad("user:1001", db); // should be hit
            cache.getOrLoad("user:9999", db); // miss -> load
            // Show TTL behaviour
            System.out.println("\n--- TTL demonstration (wait for 6 seconds) ---");
            try { Thread.sleep(6000); } catch (InterruptedException ignored) {}
            cache.getOrLoad("user:1001", db); // TTL expired -> miss and reload
            // To trigger LRU eviction for demo, you would insert many large keys,
            // exceeding maxmemory set in Redis server. Keys least recently used get evicted.
            System.out.println("\n--- LRU eviction note ---");
            System.out.println("If Redis maxmemory is small, inserting many keys causes
LRU eviction");
            System.out.println("Server eviction policy (server-side): " +
jedis.configGet("maxmemory-policy"));
            cache.printInfo();
        }
    }
}
```

Output :

Connected to Redis: PONG

--- Warm cache ---

[CACHE MISS] user:1001 -> loading from DB

[CACHE MISS] user:1002 -> loading from DB

--- Access pattern ---

[CACHE HIT] user:1001

[CACHE MISS] user:9999 -> loading from DB

--- TTL demonstration (wait for 6 seconds) ---

[CACHE MISS] user:1001 -> loading from DB

--- LRU eviction note ---

If Redis maxmemory is small, inserting many keys causes LRU eviction

Server eviction policy (server-side): allkeys-lru

Redis INFO: # Memory

Current TTL of sample key (if exists): -2

Result :

Thus the working of Redis cache for an API by explaining in-memory caching with TTL (Time-To-Live) and LRU (Least Recently Used) eviction policy was successfully designed.

Ex No : 4	Producer-Consumer system design using Kafka to demonstrate an order processing queue
Date:	

Aim:

To Simulate a producer-consumer system design using Kafka to demonstrate an order processing queue.

Procedure:

1. Start Kafka broker and create topic orders
2. Implement a Java Producer that sends JSON order messages to orders.
3. Implement a Java Consumer that listens to orders, processes each order and commits offsets (either auto or manual commit shown).
4. Simulate multiple producers and multiple consumers (consumer group) to show scaling.

Program:

OrderProducer.java

```
import org.apache.kafka.clients.producer.*;

import org.apache.kafka.common.serialization.StringSerializer;

import com.fasterxml.jackson.databind.ObjectMapper;

import java.util.Properties;

import java.util.UUID;

public class OrderProducer {

   private final KafkaProducer<String, String> producer;

   private final String topic;

   private final ObjectMapper mapper = new ObjectMapper();

   public OrderProducer(String bootstrapServers, String topic) {

      Properties props = new Properties();
```

```
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
        // durability settings
        props.put(ProducerConfig.ACKS_CONFIG, "all");
        props.put(ProducerConfig.RETRIES_CONFIG, 3);
        this.producer = new KafkaProducer<>(props);
        this.topic = topic;
    }
    public void sendOrder(Order order) throws Exception {
        String value = mapper.writeValueAsString(order);
        // key by orderId ensures ordering per key partition
        ProducerRecord<String, String> record = new ProducerRecord<>(topic,
order.orderId, value);
        producer.send(record, (metadata, exception) -> {
            if (exception != null) {
                System.err.println("Failed to send order " + order.orderId);
                exception.printStackTrace();
            } else {
                System.out.printf("Sent order %s to partition %d offset %d%n",
                        order.orderId, metadata.partition(), metadata.offset());
            }
        });
    }
```

```
public void close() {
    producer.flush();
    producer.close();
}
public static class Order {
    public String orderId;
    public String userId;
    public double amount;
    public String item;
    public long timestamp;
    public Order(String userId, String item, double amount) {
        this.orderId = UUID.randomUUID().toString();
        this.userId = userId;
        this.item = item;
        this.amount = amount;
        this.timestamp = System.currentTimeMillis();
    }
}
// Simple main to send a few orders
public static void main(String[] args) throws Exception {
    OrderProducer p = new OrderProducer("localhost:9092", "orders");
    for (int i = 0; i < 10; i++) {
        Order o = new Order("user-" + (i%3), "item-" + i, 10.5 + i);
        p.sendOrder(o);
        Thread.sleep(200);
```

```
        }
        p.close();
    }
}
```

OrderConsumer.java

```
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.TopicPartition;
import org.apache.kafka.common.serialization.StringDeserializer;
import com.fasterxml.jackson.databind.ObjectMapper;
import java.time.Duration;
import java.util.*;
public class OrderConsumer {
    private final KafkaConsumer<String, String> consumer;
    private final String topic;
    private final ObjectMapper mapper = new ObjectMapper();
    public OrderConsumer(String bootstrapServers, String topic, String groupId) {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        props.put(ConsumerConfig.GROUP_ID_CONFIG, groupId);
        // disable auto-commit to demonstrate manual commit (at-least-once behavior)
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "false");
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
```

```
        this.consumer = new KafkaConsumer<>(props);
        this.topic = topic;
    }
    public void start() {
        consumer.subscribe(Collections.singletonList(topic));
        System.out.println("Consumer started, waiting for orders...");
        try {
            while (true) {
                ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(1000));
                if (records.isEmpty()) continue;
                for (ConsumerRecord<String, String> rec : records) {
                    try {
                        OrderProducer.Order order = mapper.readValue(rec.value(),
OrderProducer.Order.class);
                        System.out.printf("Processing order %s from user %s amount=%.2f
partition=%d offset=%d%n",
                                order.orderId, order.userId, order.amount, rec.partition(), rec.offset());
                    } catch (Exception e) {
                        System.err.println("Processing failed for record, skipping commit for this
batch");
                        e.printStackTrace();
                    }
                }
                // commit offsets after batch processing (manual commit)
                try {
                    consumer.commitSync();
```

```
                System.out.println("Offsets committed");
            } catch (CommitFailedException cfe) {
                System.err.println("Commit failed: " + cfe.getMessage());
            }
        }
    } finally {
        consumer.close();
    }
  }
  public static void main(String[] args) {
    OrderConsumer c = new OrderConsumer("localhost:9092", "orders", "order-processors");
    c.start();
  }
}
```

Output :

```
Sent order 9b2b... to partition 1 offset 12
Processing order 9b2b... from user user-1 amount=11.50 partition=1 offset=12
Offsets committed
```

Result :

Thus a producer-consumer system design using Kafka to demonstrate an order processing queue was successfully designed.

Ex No : 5	Design and compare SQL and NoSQL schema for a blogging platform, highlighting the concepts of normalization and denormalization
Date:	

Aim:

To design and compare SQL and NoSQL schema for a blogging platform, highlighting the concepts of normalization and denormalization.

Procedure:

1. Identify core blog entities: Users, Posts, Comments, Tags.
2. Create relational SQL schema with foreign keys and normalized structure.
3. Create NoSQL denormalized document structure for fast reads.
4. Highlight advantages and disadvantages of both approaches.

Program:

SQL:

```
CREATE TABLE posts (
  id BIGSERIAL PRIMARY KEY,
  author_id BIGINT,
  title VARCHAR(255),
  content TEXT
);
CREATE TABLE comments (
  id BIGSERIAL PRIMARY KEY,
  post_id BIGINT,
  comment TEXT
);
```

No-SQL:

```
{
 "post_id": "p1",
 "title": "Hello World",
 "content": "First post...",
 "comments": [
   { "user": "User1", "comment": "Nice!" }
 ]
}
```

Explanation:

Normalization (SQL)

1. **Data Integrity & Consistency:** Each entity (Users, Posts, Comments, Tags) is stored separately and linked using foreign keys. This reduces redundancy and ensures consistent updates.
2. **Efficient Storage:** Since repeated information (e.g., user details in every comment) is avoided, database storage is optimized.
3. **Complex Queries:** Retrieving a blog post with its comments requires joins across multiple tables, which can slightly increase query complexity and read latency.

Denormalization (NoSQL)

1. **Improved Read Performance:** All related data (post and comments) are stored within a single document, reducing the need for joins and enabling faster reads.
2. **Redundant Data Storage:** Some data, like user details, may be repeated across documents, increasing storage usage.
3. **Simpler Reads, Complex Writes:** Updating repeated data (e.g., user info) across multiple documents can be complex but read operations remain extremely efficient.

Result :

Thus the SQL and NoSQL schema for a blogging platform, highlighting the concepts of normalization and denormalization was successfully designed.

Ex No : 6 Date:	Implementation of Circuit Breaker and Retry Logic using Resilience4j

Aim:

To implement and demonstrate Circuit Breaker and Retry logic in a microservice using Resilience4j to handle service failures gracefully and improve fault tolerance.

Procedure:

1. Define the purpose of the Circuit Breaker — to prevent repeated calls to a failing service and protect the system from cascading failures.

2. Configure the Circuit Breaker with parameters like failure rate threshold, wait duration in open state, and sliding window size.

3. Define the Retry logic configuration with parameters such as maximum retry attempts and waiting duration between retries.

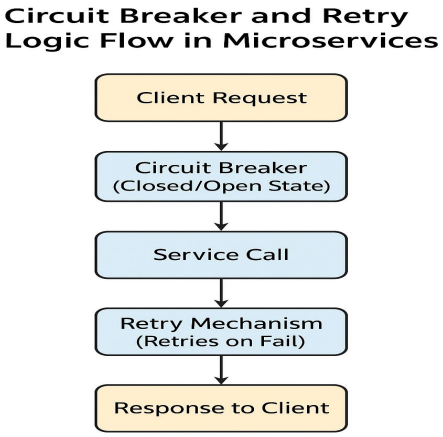
4. Create a simulated remote service call that may fail randomly to test the fault-tolerance behavior.

5. Wrap the service call using Resilience4j decorators that combine both Circuit Breaker and Retry logic.

6. Execute multiple service requests and observe how the Circuit Breaker opens after consecutive failures and retries occur automatically.

7. Verify that after the wait duration, the Circuit Breaker moves to the half-open state, allowing limited test requests before closing again upon success.

Diagram:



Program :

```
// ServiceCaller.java

import io.github.resilience4j.circuitbreaker.*;

import io.github.resilience4j.retry.*;

import io.github.resilience4j.decorators.Decorators;

import java.util.function.Supplier;

public class ServiceCaller {

   public static void main(String[] args) throws Exception {

      // Create CircuitBreaker configuration

      CircuitBreakerConfig circuitBreakerConfig = CircuitBreakerConfig.custom()

            .failureRateThreshold(50)

            .waitDurationInOpenState(java.time.Duration.ofSeconds(5))

            .slidingWindowSize(5)

            .build();

             CircuitBreaker circuitBreaker = CircuitBreaker.of("myCircuitBreaker",
circuitBreakerConfig);

     // Create Retry configuration

       RetryConfig retryConfig = RetryConfig.custom()

            .maxAttempts(3)

            .waitDuration(java.time.Duration.ofSeconds(1))

            .build();

           Retry retry = Retry.of("myRetry", retryConfig)

      // Simulated remote service call

     Supplier<String> remoteService = () -> {

        System.out.println("Attempting remote service call...");

    if (Math.random() < 0.7) { // 70% chance of failure

            throw new RuntimeException("Service failed!");

        }
```

```
            return "Success Response from Service";
        };
        // Combine CircuitBreaker and Retry
        Supplier<String> decoratedSupplier = Decorators.ofSupplier(remoteService)
                .withCircuitBreaker(circuitBreaker)
                .withRetry(retry)
                .decorate();
            for (int i = 1; i <= 10; i++) {
            try {
                String result = decoratedSupplier.get();
                System.out.println("Request " + i + ": " + result);
            } catch (Exception e) {
                System.out.println("Request " + i + ": " + e.getMessage());
            }
            Thread.sleep(1000);
        }
    }
}
```

Output :

Attempting remote service call...

Request 1: Service failed!

Attempting remote service call...

Request 2: Service failed!

Attempting remote service call...

Request 3: Service failed!

CircuitBreaker is now OPEN - blocking further calls

Request 4: CircuitBreaker 'myCircuitBreaker' is OPEN and does not permit further calls

Request 5: CircuitBreaker 'myCircuitBreaker' is OPEN and does not permit further calls

(After 5 seconds)

Attempting remote service call...

Request 6: Success Response from Service

CircuitBreaker is now CLOSED again

Request 7: Success Response from Service

Result :

Thus, the Circuit Breaker and Retry mechanisms were successfully implemented using Resilience4j in Java to improve system resilience and handle transient failures gracefully in microservice communication.

Ex No : 7	Implementation of Token Bucket Rate Limiter
Date:	

Aim:

To implement and demonstrate a Token Bucket Rate Limiter in Java to control the rate of incoming API requests.

Procedure:

1. Define the maximum capacity of the token bucket (number of tokens it can hold).
2. Define the rate at which tokens are refilled (tokens per second).
3. For each incoming request, check if a token is available in the bucket.
4. If a token is available, allow the request and consume one token.
5. If the bucket is empty, reject the request (rate limit exceeded).
6. Continuously refill tokens at the defined rate in the background.

Program :

```
// TokenBucketRateLimiter.java
public class TokenBucketRateLimiter {
   private final long capacity;
   private final long refillRatePerSec;
   private double tokens;
   private long lastRefillTimestamp;
   public TokenBucketRateLimiter(long capacity, long refillRatePerSec) {
      this.capacity = capacity;
      this.refillRatePerSec = refillRatePerSec;
      this.tokens = capacity;
      this.lastRefillTimestamp = System.nanoTime();
   }
   private void refill() {
      long now = System.nanoTime();
      double tokensToAdd = ((now - lastRefillTimestamp) / 1_000_000_000.0) *
refillRatePerSec;
```

```
        tokens = Math.min(capacity, tokens + tokensToAdd);
        lastRefillTimestamp = now;
    }
    public synchronized boolean allowRequest() {
        refill();
        if (tokens >= 1) {
            tokens--;
            return true;
        }
        return false;
    }
    public static void main(String[] args) throws InterruptedException {
        TokenBucketRateLimiter limiter = new TokenBucketRateLimiter(5, 2); // 5 tokens
max, 2/sec refill
        for (int i = 1; i <= 10; i++) {
            if (limiter.allowRequest()) {
                System.out.println("Request " + i + " allowed ");
            } else {
                System.out.println("Request " + i + " rejected  - Rate limit exceeded");
            }
            Thread.sleep(400);}
        }
    }
```

Output :

```
Request 1 allowed
Request 2 allowed
Request 3 allowed
Request 4 allowed
Request 5 allowed
Request 6 rejected  - Rate limit exceeded
Request 7 rejected  - Rate limit exceeded
Request 8 allowed
Request 9 allowed
Request 10 allowed
```

Result :

Thus, a Token Bucket Rate Limiter was successfully implemented in Java to control and monitor the rate of incoming API requests.

Ex No : 8	Failover and Heartbeat Mechanisms in a Load-Balanced Application using Mock Services
Date:	

Aim:

To simulate and understand the working of failover and heartbeat mechanisms in a load-balanced system, ensuring service availability when one or more servers fail.

Procedure:

1. Define MockService class with attributes: name and isAlive, methods: sendHeartbeat(), failService() and recoverService().
2. Cfine loadBalancer class that maintains a list of services, monitors their heartbeats, routes requests to the active(healthy) service, performs failover by switching to another service when one fails.
3. Implement the Main program; In the Failover class, create two mock services(Service A and Service B)
4. Initialize the LoadBalancer with these services, run a loop that calls checkHeartbeats() to monitor health, routeRequest() to route traffic, randomly stimulate failure and recovery events, pauses between cycles using Thread.sleep().
5. Compile and run the program, observe the console output showing normal operation, failure detection, automatic failover to the backup service, recovery and restoration of service.

Program :

```
// Failover.java
import java.util.*;

class MockService {
   String name;
   boolean isAlive = true;

   MockService(String name) {
      this.name = name;
   }

   void sendHeartbeat() {
      if (isAlive) {
         System.out.println(name + " heartbeat: ALIVE ");
      } else {
         System.out.println(name + " heartbeat: FAILED ");
      }
   }
```

```
    void failService() {
        isAlive = false;
        System.out.println(name + " has FAILED!");
    }

    void recoverService() {
        isAlive = true;
        System.out.println(name + " has RECOVERED!");
    }
}

class LoadBalancer {
    List<MockService> services;
    MockService activeService;

    LoadBalancer(List<MockService> services) {
        this.services = services;
        this.activeService = services.get(0);
    }

    void checkHeartbeats() {
        for (MockService s : services) {
            s.sendHeartbeat();
        }
    }

    void routeRequest() {
        if (!activeService.isAlive) {
            System.out.println(activeService.name + " is down. Switching to backup
service...");
            for (MockService s : services) {
                if (s.isAlive) {
                    activeService = s;
                    System.out.println("Now routing to: " + activeService.name);
                    break;
                }
            }
        } else {
            System.out.println("Routing request to: " + activeService.name);
        }
    }
}

public class Failover {
    public static void main(String[] args) throws InterruptedException {
```

```
        MockService serviceA = new MockService("Service A");
        MockService serviceB = new MockService("Service B");

        LoadBalancer lb = new LoadBalancer(Arrays.asList(serviceA, serviceB));

        for (int i = 1; i <= 6; i++) {
            System.out.println("\n--- Cycle " + i + " ---");
            lb.checkHeartbeats();
            lb.routeRequest();

            // Simulate random failure/recovery
            if (i == 3) serviceA.failService();
            if (i == 5) serviceA.recoverService();

            Thread.sleep(1000);
        }
    }
}
```

Output :

```
--- Cycle 1 ---
Service A heartbeat: ALIVE
Service B heartbeat: ALIVE
Routing request to: Service A

--- Cycle 2 ---
Service A heartbeat: ALIVE
Service B heartbeat: ALIVE
Routing request to: Service A

--- Cycle 3 ---
Service A has FAILED!
Service A heartbeat: FAILED
Service B heartbeat: ALIVE
Service A is down. Switching to backup service...
Now routing to: Service B

--- Cycle 4 ---
Service A heartbeat: FAILED
Service B heartbeat: ALIVE
Routing request to: Service B
```

--- Cycle 5 ---
Service A has RECOVERED!
Service A heartbeat: ALIVE
Service B heartbeat: ALIVE
Routing request to: Service B

--- Cycle 6 ---
Service A heartbeat: ALIVE
Service B heartbeat: ALIVE
Routing request to: Service B

Result :

Thus, a Failover and Heartbeat mechanism was successfully implemented in Java using mock services to ensure continuous availability and automatic recovery in a load-balanced application.

Ex No: 9	Implementation of Microservices for E-Commerce Checkout System using REST API
Date:	

Aim:

To design and implement microservices for an e-commerce checkout system using REST API concepts.

Procedure:

1. Define the core microservices — Cart, Payment, and Inventory.
2. Design RESTful APIs for each service with clear endpoints and data formats.
3. Create separate databases for independent operation of each service.
4. Implement inter-service communication using REST API calls.
5. Ensure error handling and data consistency during checkout.
6. Test the complete checkout process for correct integration.

Program:

```
@RestController
class CartService {
   @PostMapping("/cart/add")
   public String addToCart(String product, int quantity) {
      return quantity + " " + product + " added to cart";
   }
}
@RestController
class PaymentService {
   @PostMapping("/payment/process")
```

```
    public String processPayment(double amount) {
        return "Payment of $" + amount + " successful";
    }
}
@RestController
class InventoryService {
    @GetMapping("/inventory/check")
    public String checkStock(String product) {
        return "Stock available for " + product;
    }
}
@SpringBootApplication
public class ECommerceApp {
    public static void main(String[] args) {
        SpringApplication.run(ECommerceApp.class, args);
    }
}
```

Output:

Cart 2: Shoes added to cart
Payment: Payment of $100 successful.
Inventory: Stock available for Shoes

Result:

Microservices for e-commerce checkout were successfully implemented using REST APIs.

Ex No: 10 Date:	Implementation of Pub/Sub Notification Service using RabbitMQ

Aim:

To implement and demonstrate a **Publish/Subscribe Notification Service** in **Java** using RabbitMQ that triggers events upon user registration.

Procedure:

1. Install and configure RabbitMQ server on the local system.
2. Add RabbitMQ Java client library to the project.
3. Create a **Producer** class that publishes a message when a new user registers.
4. Create a **Consumer** class that listens to the message queue.
5. Connect both producer and consumer to the same queue using RabbitMQ connection.
6. Verify that the consumer receives the event when a user registers.

Program:

UserRegistrationProducer.java:

```
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
public class UserRegistrationProducer {
   private final static String QUEUE_NAME = "user_notifications";
   public static void main(String[] args) throws Exception {
      ConnectionFactory factory = new ConnectionFactory();
      factory.setHost("localhost");
      try (Connection connection = factory.newConnection();
           Channel channel = connection.createChannel()) {
         channel.queueDeclare(QUEUE_NAME, false, false, false, null);
         String message = "New user registered: Kirtick";
         channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
         System.out.println(" [x] Sent: " + message);
      }
   }
```

```
}
```

UserNotificationConsumer.java:

```
import com.rabbitmq.client.*;
public class UserNotificationConsumer {
   private final static String QUEUE_NAME = "user_notifications";
   public static void main(String[] argv) throws Exception {
      ConnectionFactory factory = new ConnectionFactory();
      factory.setHost("localhost");
      Connection connection = factory.newConnection();
      Channel channel = connection.createChannel();
      channel.queueDeclare(QUEUE_NAME, false, false, false, null);
      System.out.println(" [*] Waiting for messages...");
      DeliverCallback deliverCallback = (consumerTag, delivery) -> {
         String message = new String(delivery.getBody(), "UTF-8");
         System.out.println(" [x] Received notification: " + message);
      };
      channel.basicConsume(QUEUE_NAME, true, deliverCallback, consumerTag ->
      {});
   }
}
```

Output :

Producer Terminal:

```
[x] Sent: New user registered: Arun
```

Consumer Terminal:

```
[*] Waiting for messages...
[x] Received notification: New user registered: Arun
```

Result:

Thus, a Pub/Sub Notification Service was successfully implemented in Java using RabbitMQ.

Ex No : 11	Demonstration of API Gateway Architecture using Express.js or NGINX
Date:	

Aim:

To design and demonstrate an API Gateway architecture using Express.js or NGINX that routes incoming client requests to multiple backend services.

Procedure:

1. Set up an API Gateway using Express.js or NGINX to handle client requests.

2. Configure routing rules for different backend services (e.g., /user, /orders, /products).

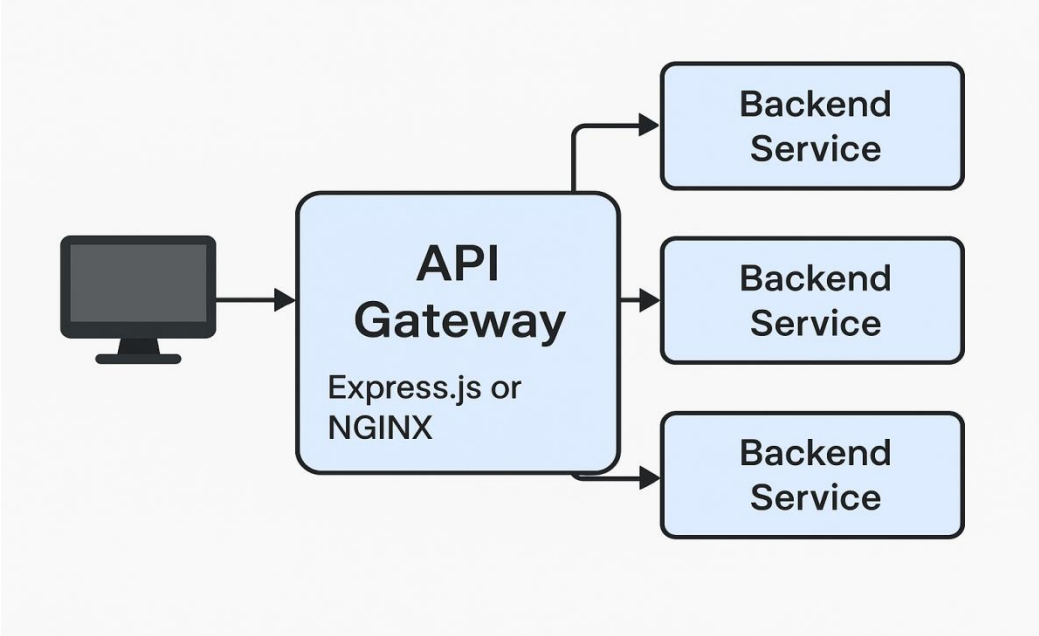
3. Forward requests to the appropriate backend service based on the route.

4. Each backend service processes the request and returns the response to the Gateway.

5. The Gateway aggregates or forwards responses back to the client.

6. Optionally, implement authentication, logging, and load balancing at the Gateway level.

Diagram:



Program :

```
// api-gateway.js
const express = require('express');
const app = express();
const PORT = 3000;

// Route to User Service
app.get('/users', (req, res) => {
  res.send('Routed to User Service');
});

// Route to Order Service
app.get('/orders', (req, res) => {
  res.send('Routed to Order Service');
});

// Route to Product Service
app.get('/products', (req, res) => {
  res.send('Routed to Product Service');
});

// Default route
app.get('/', (req, res) => {
  res.send('Welcome to API Gateway');
});
```

```
app.listen(PORT, () => {
  console.log(`API Gateway running on port ${PORT}`);
});
```

Output :

API Gateway running on port 3000

Result :

Thus, an API Gateway architecture was successfully designed using Express.js or NGINX to route client traffic to multiple backend services efficiently.

Ex No : 12	Implementation of Real-Time Chat Application
Date:	

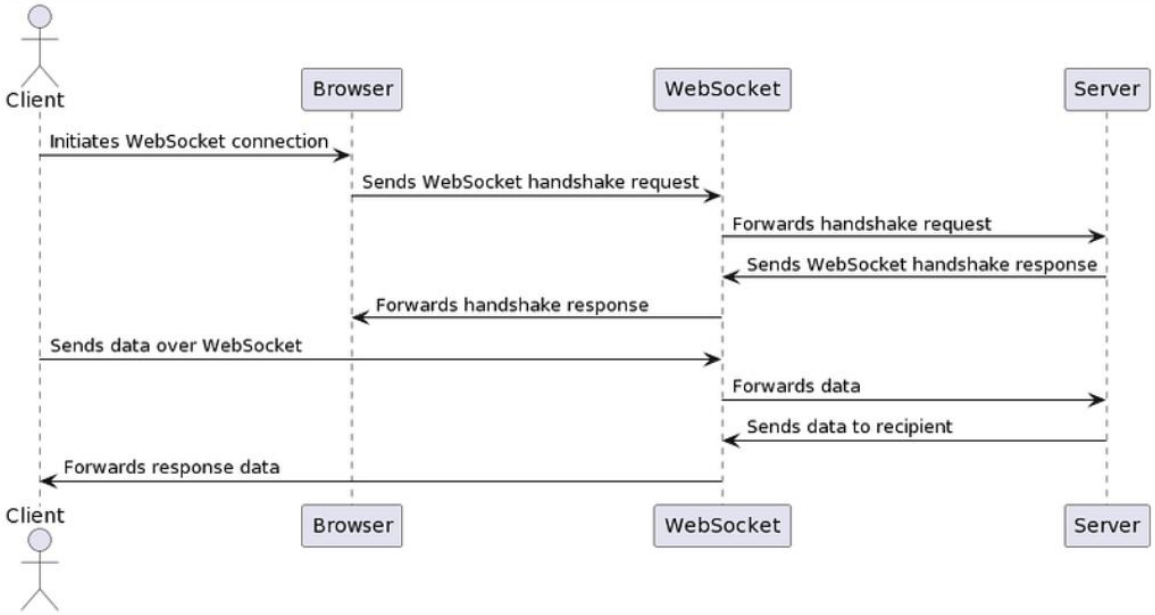
Aim:

To design a scalable chat system using WebSockets, message queues, and databases for reliable real-time communication.

Procedure:

1. Initialize Node.js and Express.js project.
2. Install dependencies: express, mongoose, socket.io, cors, dotenv, jsonwebtoken, bcryptjs.
3. Setup React frontend using create-vite or create-react-app.
4. Install React dependencies: axios, react-router-dom, socket.io-client, tailwind or material-ui.
5. Create MongoDB cluster and connect using Mongoose.
6. Define schemas for Users, Messages, and Chat Rooms.
7. Configure Express server with middleware (CORS, JSON parsing).
8. Implement REST APIs for user authentication and chat retrieval.
9. Integrate Socket.io for real-time communication.
10. Design frontend UI (Login, Signup, Chat Dashboard, Message Window, User List).
11. Connect frontend to backend using Axios and WebSockets.
12. Implement real-time message updates and event handling.
13. Test multi-browser real-time functionality.
14. Deploy frontend on Vercel/Netlify and backend on Render/Railway/AWS.

Diagram:



Result :

The design a scalable chat system using WebSockets, message queues, and databases has been successfully designed.

Ex No : 13	Implementation of servlet-based user management with CRUD operations.
Date:	

Aim:

To develop a Servlet-based User Management Application that performs CRUD (Create, Read, Update, Delete) operations on user data using Java Servlets, JSP, and JDBC.

Procedure:

1. Create a **database** userdb with table users(id, name, email, country).
2. Create a **Java Servlet project** and configure **JDBC** connection.
3. Create:
    a. User.java – Model class
    b. UserDAO.java – Handles DB CRUD operations
    c. UserServlet.java – Controls request/response
4. Create JSP pages – user-form.jsp and user-list.jsp.
5. Configure servlet in web.xml and run on Tomcat.

Program :

// User.java

```
package model;
public class User {
    private int id;
    private String name;
    private String email;
    private String country;

    public User(int id, String name, String email, String country) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.country = country;
    }

    public User(String name, String email, String country) {
        this.name = name;
```

```
        this.email = email;
        this.country = country;
    }

    // Getters and Setters
    public int getId() { return id; }
    public String getName() { return name; }
    public String getEmail() { return email; }
    public String getCountry() { return country; }

    public void setId(int id) { this.id = id; }
    public void setName(String name) { this.name = name; }
    public void setEmail(String email) { this.email = email; }
    public void setCountry(String country) { this.country = country; }
}

//UserDAO.java
import java.sql.*; import java.util.*;
public class UserDAO {
 Connection getCon() throws Exception {
  return
DriverManager.getConnection("jdbc:mysql://localhost:3306/userdb","root","password");
 }
 public void insert(User u)throws Exception{
  PreparedStatement ps=getCon().prepareStatement("insert into
users(name,email,country)values(?,?,?)");
ps.setString(1,u.name);ps.setString(2,u.email);ps.setString(3,u.country);ps.executeUpdate
();
 }
 public List<User> getAll()throws Exception{
  List<User> list=new ArrayList<>();
  ResultSet rs=getCon().createStatement().executeQuery("select * from users");
  while(rs.next())list.add(new
User(rs.getInt(1),rs.getString(2),rs.getString(3),rs.getString(4)));
  return list;
 }
}

//UserServlet.java
@WebServlet("/")
```

```
public class UserServlet extends HttpServlet {
 UserDAO dao=new UserDAO();
 protected void doGet(HttpServletRequest r,HttpServletResponse s)throws
IOException,ServletException{
  try{
   List<User> list=dao.getAll();
   r.setAttribute("list",list);
   r.getRequestDispatcher("user-list.jsp").forward(r,s);
  }catch(Exception e){e.printStackTrace();}
 }
 protected void doPost(HttpServletRequest r,HttpServletResponse s)throws IOException{
  try{dao.insert(new
User(r.getParameter("name"),r.getParameter("email"),r.getParameter("country")));
  s.sendRedirect("list");}catch(Exception e){e.printStackTrace();}
 }
}
```

Output :

1. **User List Page**

ID | Name | Email | Country | Actions

---

1 | John | john@gmail.com | India | Edit | Delete

2 | Alice | alice@yahoo.com | USA | Edit | Delete

2. **Add / Edit Form**

Name: [_____]

Email: [_____]

Country: [_____]

[Save] [Cancel]

Result :

Thus, a Servlet-based User Management Application was successfully developed and implemented in Java using JSP, Servlets, and JDBC to perform and manage CRUD operations on user data efficiently.

Ex No : 14	REST API architecture using Spring Boot or JAX-RS for a task management system.
Date:	

Aim:

To design and implement a RESTful API architecture for a Task Management System using Spring Boot.

Procedure:

1. Define models: Task, User, Project.

2. Use Spring Boot with JPA for data persistence.

3. Implement Controller → Service → Repository layers.

4. Provide CRUD operations for tasks.

5. Add a Token Bucket Rate Limiter filter for API request control.

Program :

```
//Entity

@Entity

public class Task {

   @Id @GeneratedValue

   private Long id;

   private String title;

   private String description;

   private String status;

}


//Repository

public interface TaskRepository extends JpaRepository<Task, Long> {}

//Service
```

```
@Service
public class TaskService {
    @Autowired private TaskRepository repo;

    public List<Task> getAll() { return repo.findAll(); }
    public Task create(Task t) { return repo.save(t); }
}
//Controller
@RestController
@RequestMapping("/api/tasks")
public class TaskController {
    @Autowired private TaskService service;

    @GetMapping public List<Task> getAll() { return service.getAll(); }
    @PostMapping public Task create(@RequestBody Task t) { return service.create(t); }
}
//Rate Limiter Filter
@Component
public class TokenBucketFilter extends OncePerRequestFilter {
    private int tokens = 5; // capacity

    @Override
    protected void doFilterInternal(HttpServletRequest req, HttpServletResponse res,
FilterChain chain)
            throws IOException, ServletException {
```

```
        if (tokens > 0) { tokens--; chain.doFilter(req, res); }
        else { res.setStatus(429); res.getWriter().write("Rate limit exceeded"); }
    }
}
```

Output :

GET /api/tasks

```
[
  { "id": 1, "title": "Design API", "description": "Prepare task endpoints", "status": "TODO" }
]
```

POST /api/tasks

```
{ "id": 2, "title": "Implement service", "description": "Add business logic", "status": "IN_PROGRESS" }
```

Result :

Thus, A simple REST API for task management is successfully designed and implemented using Spring Boot, supporting CRUD operations and protected by a Token Bucket Rate Limiter.

Ex No : 15	Implementation of Capstone System Design
Date:	

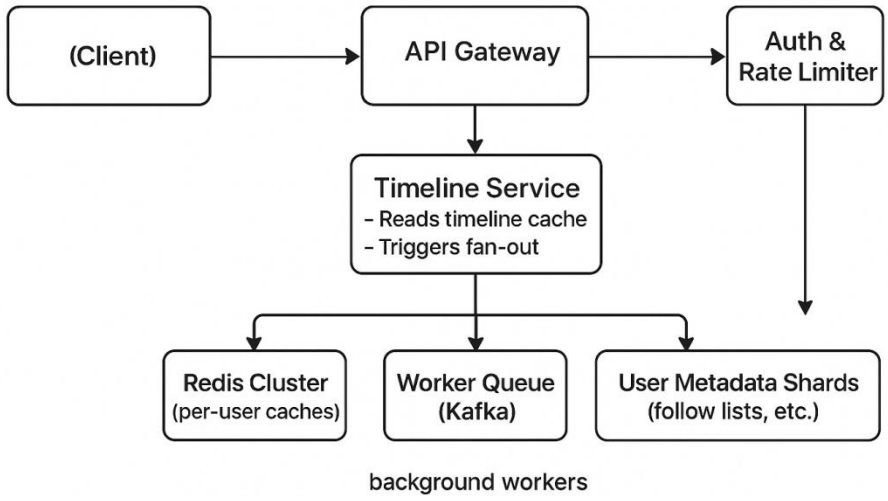
Aim:

To design a scalable Twitter-like platform focusing on database sharding, feed generation, and caching to efficiently handle millions of users and real-time data.

Procedure:

1. Divide user data across multiple **database shards** using a hash-based sharding strategy.
2. Implement **feed generation** using a hybrid approach — fan-out-on-write for regular users and fan-out-on-read for celebrity users.
3. Use **Redis caching** to store recent tweets and timelines for quick retrieval.
4. Employ **message queues** (e.g., Kafka) for background fan-out and timeline updates.
5. Use an **API Gateway** for handling user requests and directing them to appropriate services.

Diagram:



Result :

Thus A Twitter-like system was successfully designed that uses database sharding for scalability, Redis caching for fast timeline retrieval, and hybrid feed generation for efficient performance and reduced latency.