Java Institute for Advanced Technology

BUSINESS COMPONENT DEVELOPMENT I
JIAT/BCD I
JIAT/BCD I/EX/01

OSHADHA KAVINTHA
SCN NO
200027703100
GAMPAHA

Task 1

01.

The Urban Traffic Management System scenario involves addressing the challenges of urban traffic congestion and pollution by implementing a smart solution. Key components of this scenario include simulating IoT devices to capture traffic-related data, setting up a central server for data processing and analysis. And another requirement is use JMS To parse capturing data to Traffic management System from IoT, to do that publisher/subscriber messaging method will be most suitable. According to requirement Traffic Management System have to work with decentralised IOTs and sensors like traffic sensors, colour light system and ect. so it is suitable to use EJB to develop the application instead of normal SE or web application. An analytical server class needs to be created to receive data from the central server and perform analysis tasks. This includes calculating average vehicle speed, identifying traffic patterns, and analysing overall urban mobility efficiency. Within the analytical server class, methods should be defined to monitor and calculate specific metrics based on the received sensor data. These metrics may include average vehicle speed, traffic flow analysis, and overall urban mobility efficiency.

challenges-

The solution needs to handle a potentially large volume of data from numerous IoT devices distributed across the urban area. The central server and analytical server should be capable of processing data in real-time to provide timely insights for traffic management decisions. To do that we can use JMS like pub/sub messaging method. Integrating different components of the system, such as the IoT device simulation, central server, and analytical server, may pose challenges in terms of system integration and interoperability.

02.

Design of the EJB Solution

Architectural Design-

The solution follows a layered architecture consisting of the following components

- IoT Simulation Layer-
  Responsible for simulating IoT devices installed in vehicles and at traffic intersections, implemented in the IOTSimulation Java SE application and contains the IOTDataPublisher and IOTDevice classes.
- EJB Layer-
  Responsible for core layer of the solution, handling data transmission, analysis, and

processing, implemented in the ejb module of the UrbanTrafficManagementSystem application and Includes components such as IOTDataBean, TrafficDataAnalyseBean, and IOTDataSubscriber.

- Web Layer-
  Responsible for provides a user interface for monitoring traffic data and analysis results, implemented in the web module of the UrbanTrafficManagementSystem application and contains servlets such as IOTDataManager and supporting files like JSP pages and client-side scripts.

Component Design-

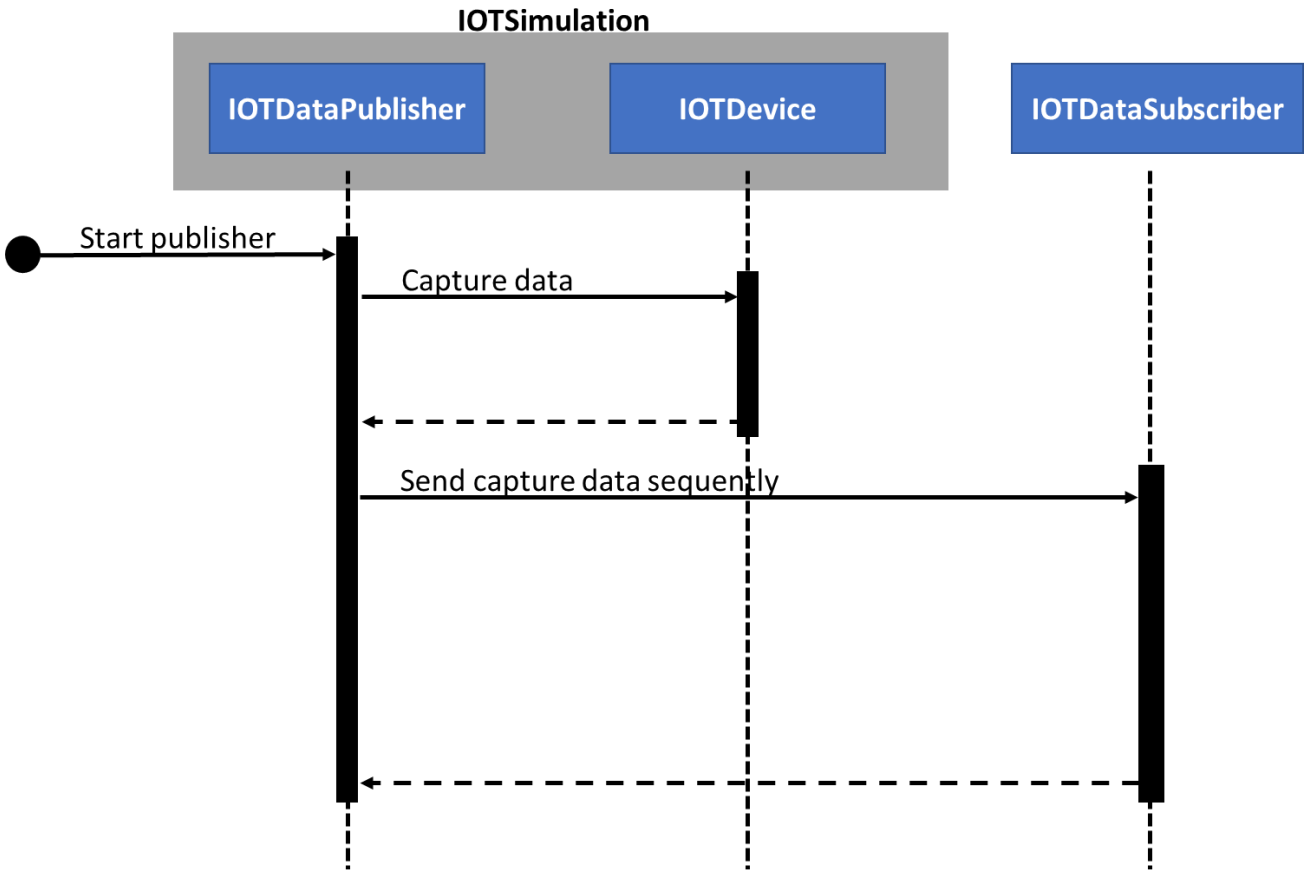
- IOTDataBean-
  It is singleton EJB and responsible for receiving IoT data from CentralServer and send it to the IOTDataManager in web module. The reason of using singleton EJB for this task it is use only one instance in EJB container and it is more reliable for this operation than stateless and stateful EJB.
- TrafficDataAnalyseBean-
  It is stateless EJB and responsible for analyzing traffic data received from AnalyticalServer and send it to the web module. Stateless EJB use instance pool in EJB container to response to the clients, because of that it would be enhance efficiency for this process than stateful and singleton.
- IOTDataSubscriber-
  Message-driven bean (MDB) responsible for receiving data from simulated IoT devices. Transfers received data to CentralServer for storage and further analysis. In this process publisher/subscriber method used for data received because it provides decoupling of publisher and subscribers, scalability, flexibility, durability and loose Coupling. But if we use queue (point to point messaging) instead of topic it can be occur problems like limited fan-out, ordering guarantees issues across multiple queues and complexity for simple Tasks.
- AnalyticalServer-
  This class responsible for coordinating data analysis tasks and interacts with TrafficDataAnalyseBean to perform analysis and generate insights and also it implements methods to calculate average vehicle speed, identify traffic patterns, and analyze urban mobility efficiency.
- CentralServer-
  This class representing the central server responsible for data reception and storage and Coordinates with IOTDataBean to store incoming data for analysis.

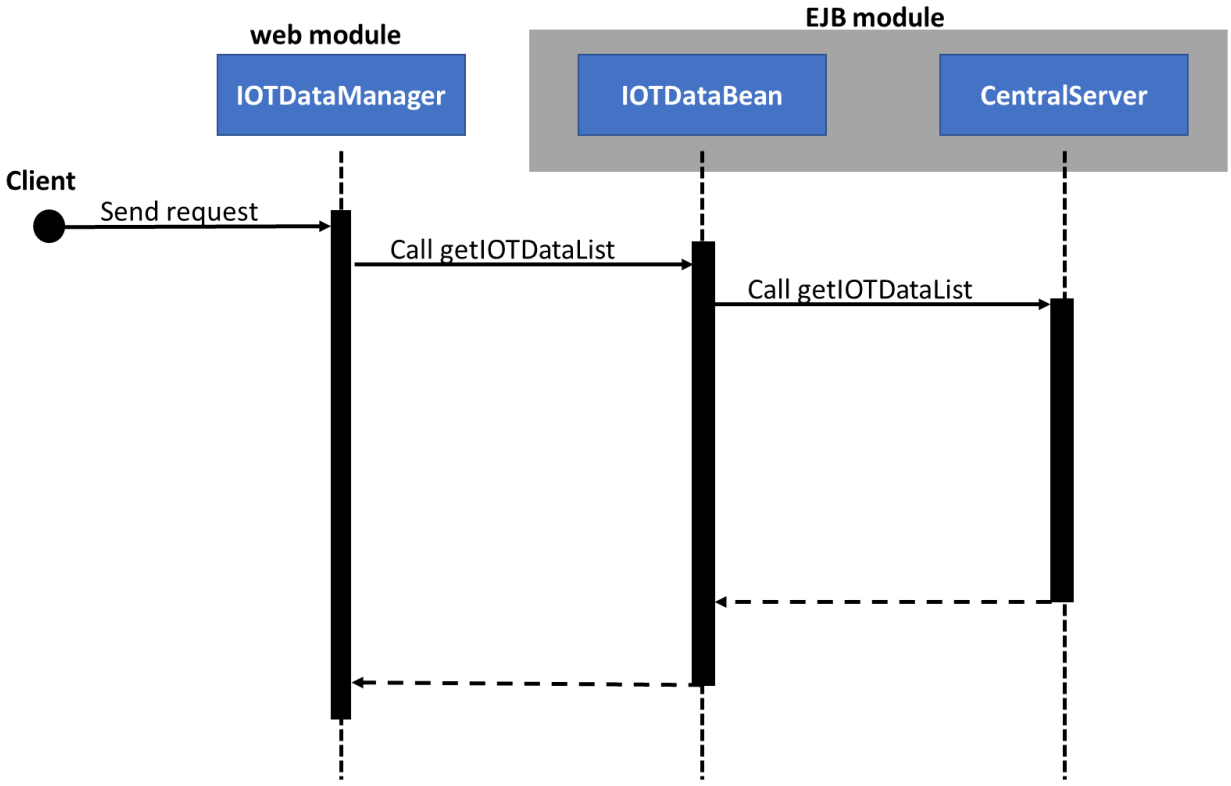
Functional Design-

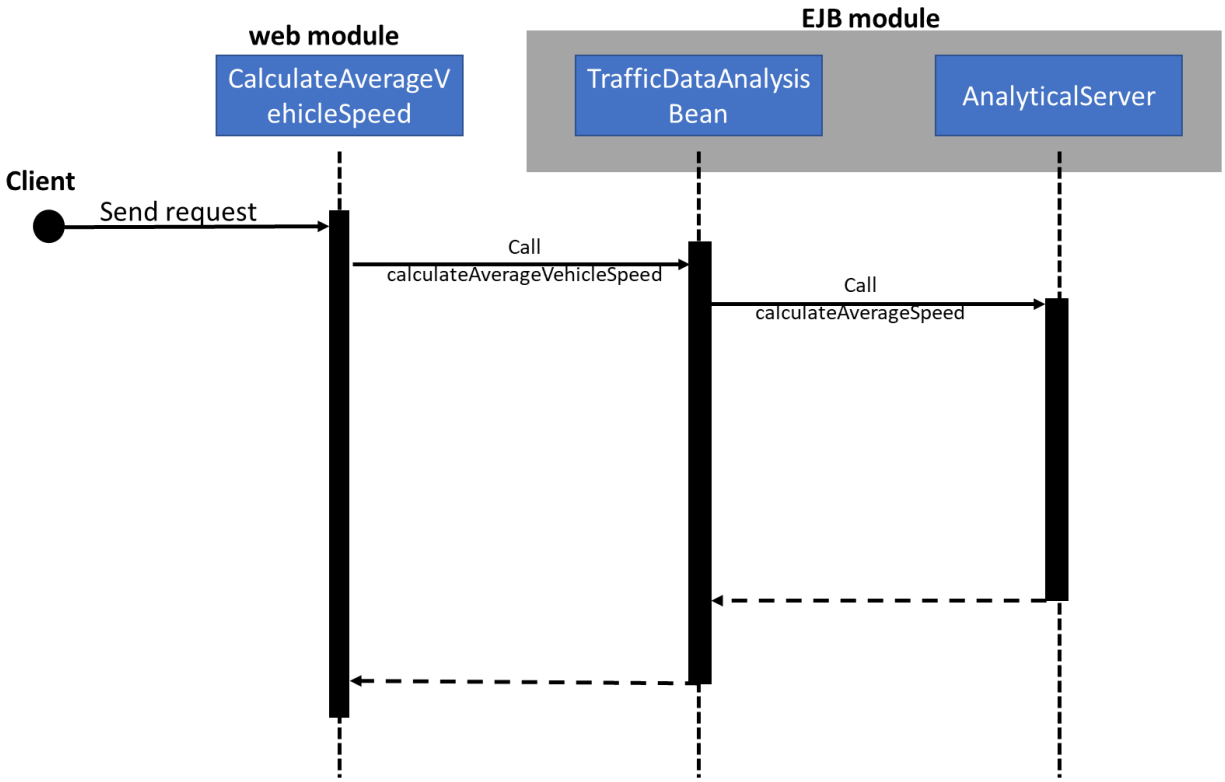
- IOTDataBean-
  The IOTDataBean receives data from the IOTDataSubscriber message-driven bean, storing it for further analysis. It then sends this data to the IOTDataManager servlet in the web module for user presentation. This component acts as a central repository for storing data received from simulated IoT devices, ensuring its persistence and availability for analysis and presentation. It utilizes a singleton EJB to maintain consistency and reliability in data storage operations within the EJB container.
- TrafficDataAnalyseBean-
  It sends these analysis results to the IOTDataManager servlet for presentation.This bean conducts advanced analysis on traffic data to extract meaningful insights, utilizing stateless EJB for efficient handling of analysis tasks. It leverages the EJB container's instance pool for scalability and responsiveness, providing real-time analysis results to the web interface.
- IOTDataSubscriber-
  The IOTDataSubscriber listens for messages from simulated IoT devices using the publisher/subscriber method. It transfers received data to the CentralServer for storage and further analysis. Serving as the entry point for incoming IoT data, this component ensures decoupling between IoT devices and the central server. It utilizes the publisher/subscriber method for scalable, flexible, and loosely coupled message communication, ensuring reliable and efficient data transfer for processing and analysis.
- AnalyticalServer-
  The AnalyticalServer coordinates data analysis tasks and interacts with the TrafficDataAnalyseBean. It implements methods to calculate average vehicle speed, identify traffic patterns, and analyze urban mobility efficiency. Acting as an intermediary between the central server and analysis components, this server manages data flow and task coordination. It provides a centralized location for implementing analysis algorithms, facilitating communication between the central server and analysis components for seamless integration and data exchange.
- CentralServer-
  The CentralServer serves as the central server for data reception and storage. It collaborates with the IOTDataBean to store incoming data for analysis. Functioning as the backbone of the system, this server receives, stores, and manages IoT data, ensuring its integrity, reliability, and availability. It closely collaborates with analysis components to support data processing and decision-making, thereby contributing to the overall goals of the traffic management system.

03.

Sequence Diagram


IOTSimulation
IOTDataPublisher
IOTDevice
IOTDataSubscriber
Start publisher
Capture data
Send capture data sequently


web module
IOTDataManager
EJB module
IOTDataBean
CentralServer
Client
Send request
Call getIOTDataList
Call getIOTDataList



Task 2

01.

The J2EE platform offers a robust framework for developing internet or intranet-based applications with exceptional scalability and availability. This evaluation assesses how the design and implementation of a traffic management system benefit from leveraging the J2EE platform's features, focusing on the Enterprise Application Model, containers and connectors, and the Enterprise JavaBeans (EJB) component model.

If consider real world example of usage of Enterprise Application Model, In the traffic management system, the Enterprise Application Model organizes components into logical layers, facilitating modular design and scalability. For instance, the IoT simulation layer simulates IoT devices capturing traffic data. The EJB layer processes and analyzes this data, while the web layer provides a user interface for monitoring and analysis.

Code Example:

- IOTData interface

```
@Remote
public interface IOTData {
   public void addIOTData(JSONObject jsonObject);
   public List<JSONObject> getIOTDataList();
}
```

- IOTDataBean class

```
@Singleton(mappedName = "api/v1/impl/IOTDataBean")
public class IOTDataBean implements IOTData {

   public void addIOTData(JSONObject jsonObject){
      CentralServer.addIOTData(jsonObject);
   }
   public List<JSONObject> getIOTDataList(){
      return CentralServer.getIOTDataList();
   }
}
```

- IOTDataManager servlet

```
@WebServlet(name = "IOTDataManager", value = "/data-view")

public class IOTDataManager extends HttpServlet {

   @Override

   protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {

      try {

         resp.setContentType("application/json");

         resp.setCharacterEncoding("UTF-8");

         InitialContext context = new InitialContext();

         IOTData iotData = (IOTData) context.lookup("api/v1/impl/IOTDataBean");

         List<JSONObject> iotDataList = iotData.getIOTDataList();

         JSONArray jsonArray = new JSONArray(iotDataList);

         resp.getWriter().write(jsonArray.toString(4));

      } catch (NamingException e) {

         throw new RuntimeException(e);

      }

   }

}
```

02.

Usage of Containers and Connectors are-

Containers provide runtime environments for EJBs, managing their lifecycle and resource allocation. Connectors facilitate communication between components, enhancing scalability by allowing distributed deployment and availability by enabling fault-tolerant connections. Payara

Server supports clustering, which enhances scalability and availability by enabling the deployment of EJBs across multiple nodes with redundancy.

If consider features enhancing scalability one feature is Container-managed concurrency. It allows multiple instances of EJBs to handle concurrent requests efficiently. Another feature is Load balancing. Distributes incoming traffic across multiple instances of EJBs, improving scalability and also clustering that enables the deployment of EJBs across multiple nodes, enhancing availability through redundancy.

- xml configuration file

```
<!-- clustering configuration in payara server -->
<ejb-container>
  <ejb-timer-service>
    <!-- enable clustering for Message-Driven Beans -->
    <mdb-container-config>
      <mdb-shared-store>true</mdb-shared-store>
      <mdb-is-clustered>true</mdb-is-clustered>
    </mdb-container-config>
  </ejb-timer-service>
</ejb-container>
```

<mdb-shared-store> enables shared storage for Message-Driven Beans (MDBs), ensuring that message processing state is shared across cluster nodes.

<mdb-is-clustered> indicates that MDBs are clustered, allowing messages to be distributed and processed by multiple instances of MDBs running on different nodes.

03.

One of the advantages of Enterprise JavaBeans Component Model is scalability. EJBs support distributed deployment, allowing components to scale horizontally across multiple servers to handle increasing workloads and also availability. EJB containers provide services such as transaction management and security, ensuring reliable and fault-tolerant execution of business logic.

examples-

IOTDataBean utilizes singleton EJB to manage data storage, ensuring a single, reliable instance for handling incoming IoT data.

TrafficDataAnalyseBean utilizes stateless EJB for analysis tasks, allowing the container to manage instances dynamically based on demand, enhancing scalability.

- IOTDataBean class

```
@Singleton(mappedName = "api/v1/impl/IOTDataBean")
public class IOTDataBean implements IOTData {
   public void addIOTData(JSONObject jsonObject){
      CentralServer.addIOTData(jsonObject);
   }
   public List<JSONObject> getIOTDataList(){
      return CentralServer.getIOTDataList();
   }
}
```

- TrafficDataAnalyseBean class

```
@Stateless(mappedName = "api/v1/impl/TrafficDataAnalyseBean")
public class TrafficDataAnalyseBean implements TrafficDataAnalyse {
   private final AnalyticalServer analyticalServer = new AnalyticalServer();

   public double calculateAverageVehicleSpeed(){
      return analyticalServer.calculateAverageSpeed(CentralServer.getIOTDataList());
   }

   public String identifyTrafficPatterns(){
      return analyticalServer.identifyTrafficPatterns(CentralServer.getIOTDataList());
   }
   public Map<String, Double> trafficFlowAnalysis(){
      return analyticalServer.performTrafficFlowAnalysis(CentralServer.getIOTDataList());
   }
   public double calculateUrbanMobilityEfficiency(){
      return analyticalServer.calculateUrbanMobilityEfficiency(CentralServer.getIOTDataList());
   }
}
```

The J2EE platform, with its Enterprise Application Model, containers and connectors, and EJB component model, provides a comprehensive framework for developing scalable and available internet or intranet-based applications like the traffic management system.

Task 3

01.

Dependency Injection

Impact on Code Maintainability and Extensibility.

- Dependency Injection (DI) simplifies code maintenance and extensibility by decoupling components and providing flexibility in swapping dependencies. In the solution, Dependency injection is evident in EJB injection, such as injecting IOTDataBean into IOTDataSubscriber. This allows easy substitution of implementations without modifying client code, enhancing maintainability.

➢ IOTDataSubscriber class

```
@MessageDriven( activationConfig = { @ActivationConfigProperty(propertyName = "destinationLookup",
propertyValue = "myTopic") } )

public class IOTDataSubscriber implements MessageListener {

   private final IOTData iotData = new IOTDataBean();

   @Override

   public void onMessage(Message message) {

      try {

         if (message instanceof TextMessage) {

            TextMessage textMessage = (TextMessage) message;

            String jsonStr = textMessage.getText();

            JSONObject jsonObject = new JSONObject(jsonStr);

            iotData.addIOTData(jsonObject);

         } else {

            System.out.println("Received message of unexpected type: " + message.getClass().getName());

         }

      } catch (JMSException e) {

         throw new RuntimeException(e);

      } }}
```

Role of Initial Context Lookup in Resource Management.

- Initial Context Lookup introduces coupling with JNDI, affecting scalability and configuration. While it provides access to resources, it can be resource-intensive and tightly binds components to the lookup process. This can hinder scalability due to potential bottlenecks in resource retrieval and configuration management. As example CalculateAverageVehicle servlet provide access to the TrafficDataAnalyseBean through JNDI.

- CalculateAverageVehicleSpeed servlet

```
@WebServlet(name = "CalculateAverageVehicleSpeed",value = "/average-speed")
public class CalculateAverageVehicleSpeed extends HttpServlet {
   @Override
   protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
      try {
         resp.setContentType("application/json");
         resp.setCharacterEncoding("UTF-8");
         InitialContext context = new InitialContext();
         TrafficDataAnalyse trafficDataAnalyse = (TrafficDataAnalyse)
               context.lookup("api/v1/impl/TrafficDataAnalyseBean");
         double averageVehicleSpeed = trafficDataAnalyse.calculateAverageVehicleSpeed();
         resp.getWriter().write(Double.toString(averageVehicleSpeed));
      } catch (NamingException e) {
         throw new RuntimeException(e);
      }
   }
}
```

- TrafficDataAnalyseBean class

```
@Stateless(mappedName = "api/v1/impl/TrafficDataAnalyseBean")
public class TrafficDataAnalyseBean implements TrafficDataAnalyse {
   private final AnalyticalServer analyticalServer = new AnalyticalServer();
   public double calculateAverageVehicleSpeed(){
      return analyticalServer.calculateAverageSpeed(CentralServer.getIOTDataList());
   }
   public String identifyTrafficPatterns(){
      return analyticalServer.identifyTrafficPatterns(CentralServer.getIOTDataList());
   }
   public Map<String, Double> trafficFlowAnalysis(){
      return analyticalServer.performTrafficFlowAnalysis(CentralServer.getIOTDataList());
   }
   public double calculateUrbanMobilityEfficiency(){
      return analyticalServer.calculateUrbanMobilityEfficiency(CentralServer.getIOTDataList());
   }
}
```

Comparison of Dependency Injection and Initial Context Lookup:

- Dependency Injection promotes loose coupling and is more suitable for modular and extensible architectures like the Smart Urban Traffic Management solution. It enhances maintainability by reducing dependencies between components. In contrast, Initial Context Lookup introduces tighter coupling and can complicate scalability and configuration management, making dependency injection the preferred approach in this scenario.
- If we consider CalculateAverageVehicleSpeed servlet it has tighter coupling with TrafficDataAnalyseBean but TrafficDataAnalyseBean has dependency injection with

Analytical server which more loosely coupled.

02.

JMS API

Benefits of JMS API

- The JMS API facilitates robust communication by providing asynchronous messaging capabilities, ensuring reliable message delivery and fault tolerance. In the solution, JMS enables communication between IoT devices and the central server. For example, the IOTDataPublisher uses JMS to transmit data to the IOTDataSubscriber, enhancing communication reliability.

➢ IOTDataPublisher class

```
public class IOTDataPublisher {

   public static void main(String[] args) {

      try {

         InitialContext context = new InitialContext();

         TopicConnectionFactory factory= (TopicConnectionFactory)
context.lookup("myTopicConnectionFactory");

         TopicConnection connection = factory.createTopicConnection();

         TopicSession session = connection.createTopicSession(false,
Session.AUTO_ACKNOWLEDGE);

         Topic topic = (Topic) context.lookup("myTopic");

         TopicPublisher publisher = session.createPublisher(topic);

         IOTDevice iotDevice;

         while (true) {

            iotDevice = new IOTDevice();

            String data = iotDevice.captureData();

            TextMessage message = session.createTextMessage();

            message.setText(data);

            publisher.publish(message);

            Thread.sleep(2000);

         }

      } catch (NamingException e) {

         throw new RuntimeException(e);

      } catch (JMSException e) {

         throw new RuntimeException(e);

      } catch (InterruptedException e) {

         throw new RuntimeException(e);

      }}}
```

- IOTDataSubscriber class

```
@MessageDriven( activationConfig = { @ActivationConfigProperty(propertyName = "destinationLookup",
propertyValue = "myTopic") } )

public class IOTDataSubscriber implements MessageListener {

   private final IOTData iotData = new IOTDataBean();

   @Override

   public void onMessage(Message message) {

      try {

         if (message instanceof TextMessage) {

            TextMessage textMessage = (TextMessage) message;

            String jsonStr = textMessage.getText();

            JSONObject jsonObject = new JSONObject(jsonStr);

            iotData.addIOTData(jsonObject);

         } else {

            System.out.println("Received message of unexpected type: " + message.getClass().getName());

         }

      } catch (JMSException e) {

         throw new RuntimeException(e);

      } }}
```

Key Use Cases of JMS in Facilitating Messaging.

- JMS plays a vital role in scenarios requiring asynchronous communication, such as real-time data transmission in the traffic management system. For instance, JMS enables efficient handling of incoming traffic data from IoT devices, ensuring timely processing and analysis by the system.

Integration of JMS API into the Solution.

- The JMS API seamlessly integrates into the solution, providing a standardized messaging interface for communication between components. Challenges may arise in configuring JMS resources and handling message processing, but proper integration ensures efficient communication. Code snippets, such as JMS message publishing and consumption in IOTDataPublisher and IOTDataSubscriber, demonstrate JMS usage in the application.

03.

Message-driven beans (MDB)

Integration of MDB for Asynchronous Message Processing.

- Message-driven beans (MDBs) are integrated into the solution to handle asynchronous message processing efficiently. For example, the IOTDataSubscriber MDB listens for messages from IoT devices and forwards them to the IOTDataBean for storage and analysis.

Role of MDB in Achieving Efficient Asynchronous Message Processing.

- MDBs enable scalable and efficient asynchronous message processing by offloading tasks to separate, concurrent threads. This ensures that message processing does not block the main application thread, enhancing system responsiveness and throughput.

Benefits of Using MDB in the Scenario.

- MDBs contribute to message handling efficiency by providing a lightweight, container-managed approach to asynchronous processing. They simplify message consumption and processing, leading to improved system performance and reliability. If get example to illustrate MDB usage, such as message consumption in IOTDataSubscriber, highlight their role in the solution.

Task 4

01.

Scalability and Availability.

A Singleton Session Bean ensures that only one instance exists per application, making it suitable for managing shared resources like data storage and analysis tasks. In the traffic management system, the IOTDataBean is implemented as a Singleton Session Bean to handle IoT data storage. This ensures consistent access to the central data repository, enhancing scalability by avoiding resource contention and enabling efficient utilization of server resources. Additionally, by centralizing data management, the Singleton Session Bean enhances availability by providing a single, reliable point of access for data storage and retrieval.

02.

Advantages of Singleton Session Bean.

Compared to Stateful and Stateless Session Beans, a Singleton Session Bean offers several advantages. It maintains state across multiple client invocations, providing consistency in data access and processing. Unlike Stateless Session Beans, it retains state between method calls. Additionally, it avoids the overhead of creating and destroying instances per client request, improving performance and resource utilization.

03.

In the traffic management system, the IOTDataBean Singleton Session Bean efficiently handles incoming IoT data, ensuring consistent storage and retrieval. For instance, when multiple IoT devices simultaneously transmit data, the Singleton Session Bean coordinates data storage and analysis tasks, preventing conflicts and ensuring data integrity. This centralized approach streamlines data management and enhances system responsiveness, ultimately improving traffic monitoring and analysis capabilities. Additionally, by maintaining a single instance per application, the Singleton Session Bean optimizes resource usage, contributing to the overall scalability and availability of the system.