

# Architecture of Enterprise Applications 2

## Messaging - JMS

Haopeng Chen

***RE***liable, ***IN***telligent and ***SC***alable Systems Group (***REINS***)

Shanghai Jiao Tong University

Shanghai, China

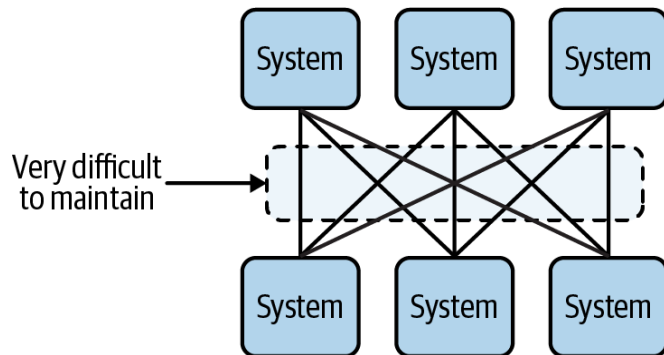
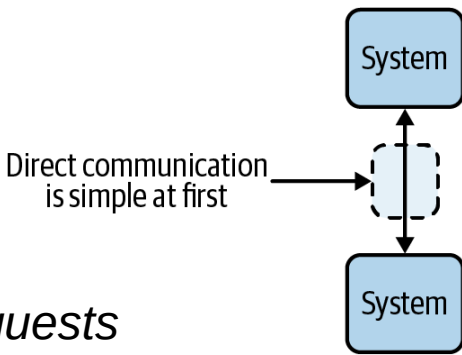
<http://reins.se.sjtu.edu.cn/~chenhp>

e-mail: chen-hp@sjtu.edu.cn

- Contents
  - Messaging in Java EE Applications
    - What is a Messaging?
    - What is JMS API?
    - JMS Programming Model
- Objective
  - 能够根据系统需求，分析后端适用于异步通信机制的业务场景，并设计并实现基于消息中间件的实现方案

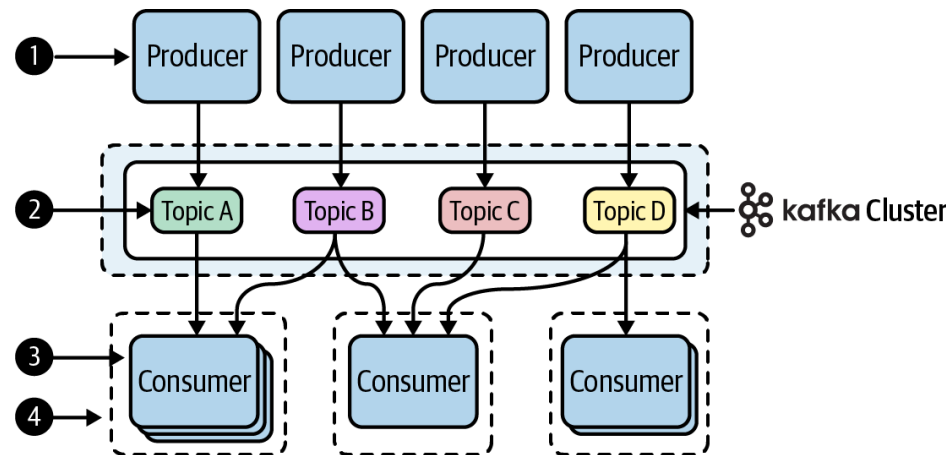
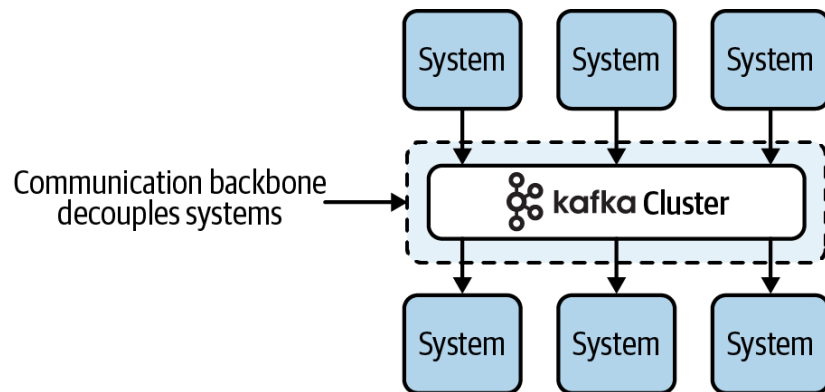
- Synchronous client-server model

- *tightly coupled*
- *no delivery guarantees*
- *software speciation*
- *without a request buffer*
- too much emphasis on *requests* and *responses*
- Communication is *not replayable*



From:  
Mastering Kafka Streams and ksqlDB - *Building Real-Time Data Systems by Example*  
Mitch Seymour

- Asynchronous model

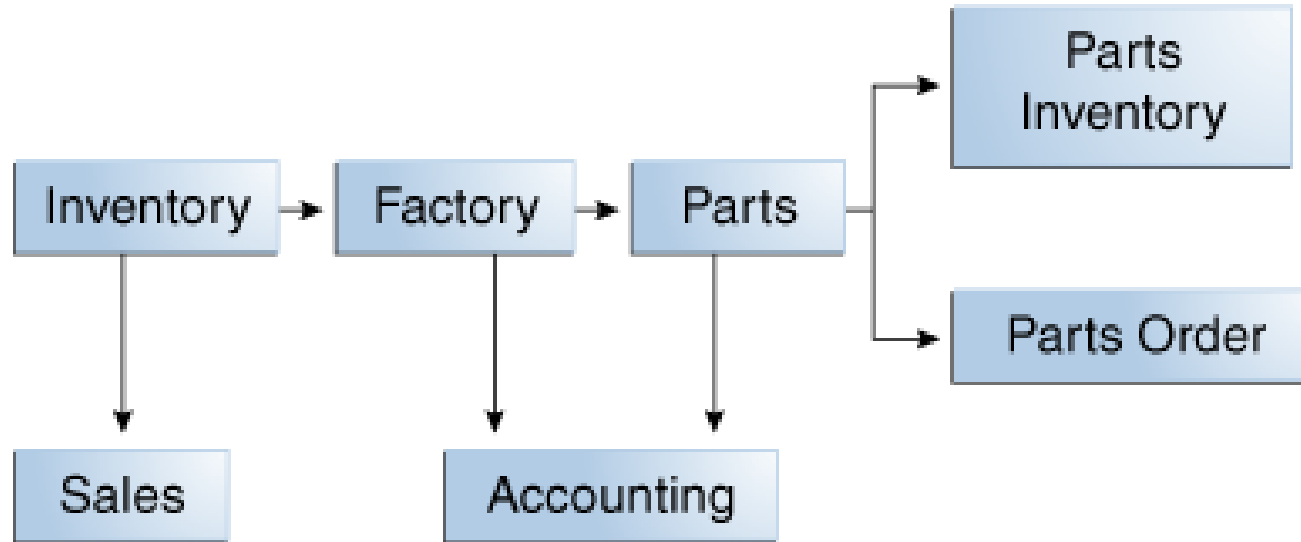


From:  
**Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example**  
*Mitch Seymour*

- Messaging is a method of communication between software components or applications.
  - A messaging system is a peer-to-peer facility:
  - A messaging client can send messages to, and receive messages from, any other client.
  - Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages.
- Messaging enables distributed communication that is **loosely coupled**.

- The Java Message Service is a Java API that allows applications to create, send, receive, and read messages.
- JMS enables communication that is not only loosely coupled but also:
  - **Asynchronous**: A receiving client does not have to receive messages at the same time the sending client sends them. The sending client can send them and go on to other tasks; the receiving client can receive them much later.
  - **Reliable**: A messaging provider that implements the JMS API can ensure that a message is delivered once and only once. Lower levels of reliability are available for applications that can afford to miss messages or to receive duplicate messages.

# When Can We Use Messaging?



- The JMS API in the Java EE platform has the following features.
  - **Application clients**, **Enterprise JavaBeans (EJB) components**, and **web components** can send or synchronously receive a JMS message. Application clients can in addition set a message listener that allows JMS messages to be delivered to it asynchronously by being notified when a message is available.
  - **Message-driven beans**, which are a kind of enterprise bean, enable the asynchronous consumption of messages in the EJB container. An application server typically pools message-driven beans to implement concurrent processing of messages.
  - **Message send and receive operations** can participate in Java Transaction API (JTA) transactions, which allow JMS operations and database accesses to take place within a single transaction.

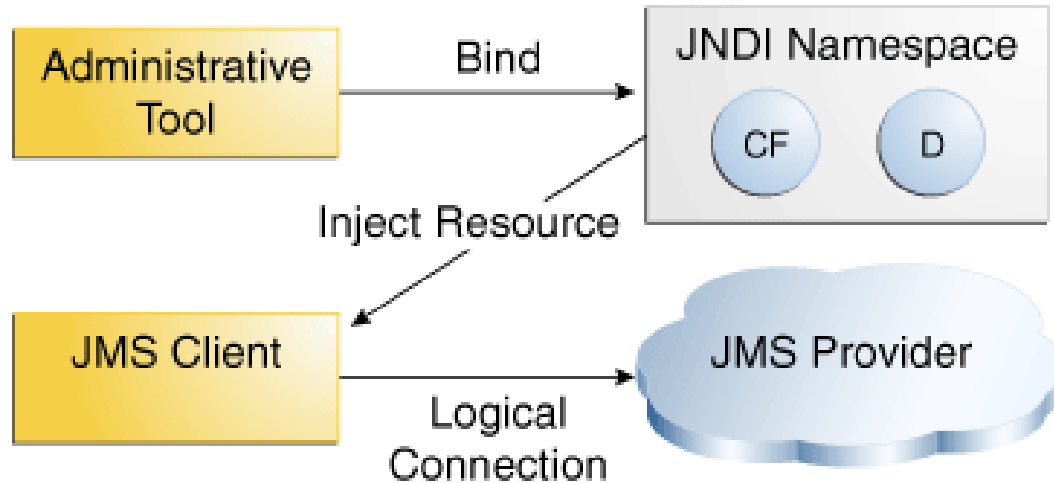


- Applications produce or consume messages
- The format of message is elastic, including three parts:
  - A header
  - Properties (optional)
  - A body (optional)

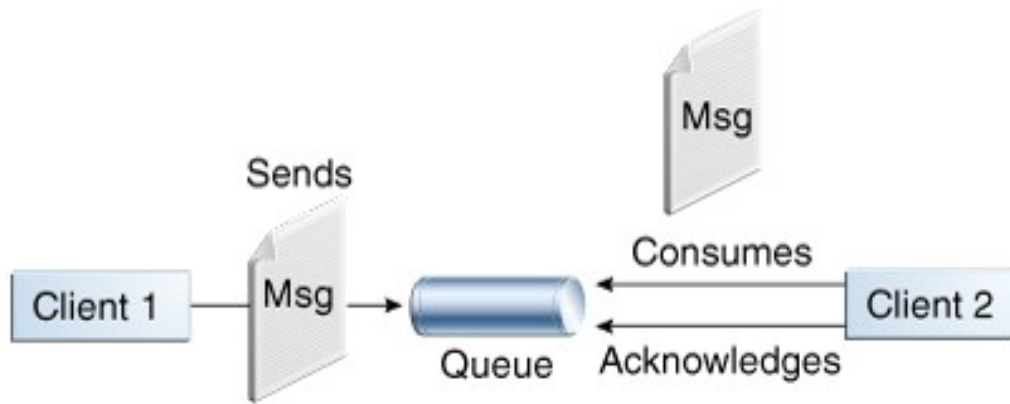
- Includes some pre-defined fields
  - JMSDestination (S)
  - JMSDeliveryMode (S)
  - JMSMessageID (S)
  - JMSTimestamp (S)
  - JMSCorrelationID (C)
  - JMSReplyTo (C)
  - JMSRedelivered (P)
  - JMSType (C)
  - JMSExpiration (S)
  - JMSPriority (S)
- Clients can not extend the fields

- Includes some pre-defined fields
  - JMSXUserID (S)
  - JMSXAppID (S)
  - JMSXDeliveryCount (S)
  - JMSXGroupID (C)
  - JMSXGroupSeq (C)
  - JMSXProducerTXID (S)
  - JMSXConsumerTXID (S)
  - JMSXRcvTimestamp (S)
  - JMSXState (P)
- Clients can extend the fields
  - Property name: follows the rule of naming selectors
  - Property value: boolean, byte, short, int, long, float, double, String

Message Type	Body Contains
TextMessage	A java.lang.String object (for example, the contents of an XML file).
MapMessage	A set of name-value pairs, with names as String objects and values as primitive types in the Java programming language. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined.
BytesMessage	A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.
StreamMessage	A stream of primitive values in the Java programming language, filled and read sequentially.
ObjectMessage	A Serializable object in the Java programming language.
Message	Nothing. Composed of header fields and properties only. This message type is useful when a message body is not required.



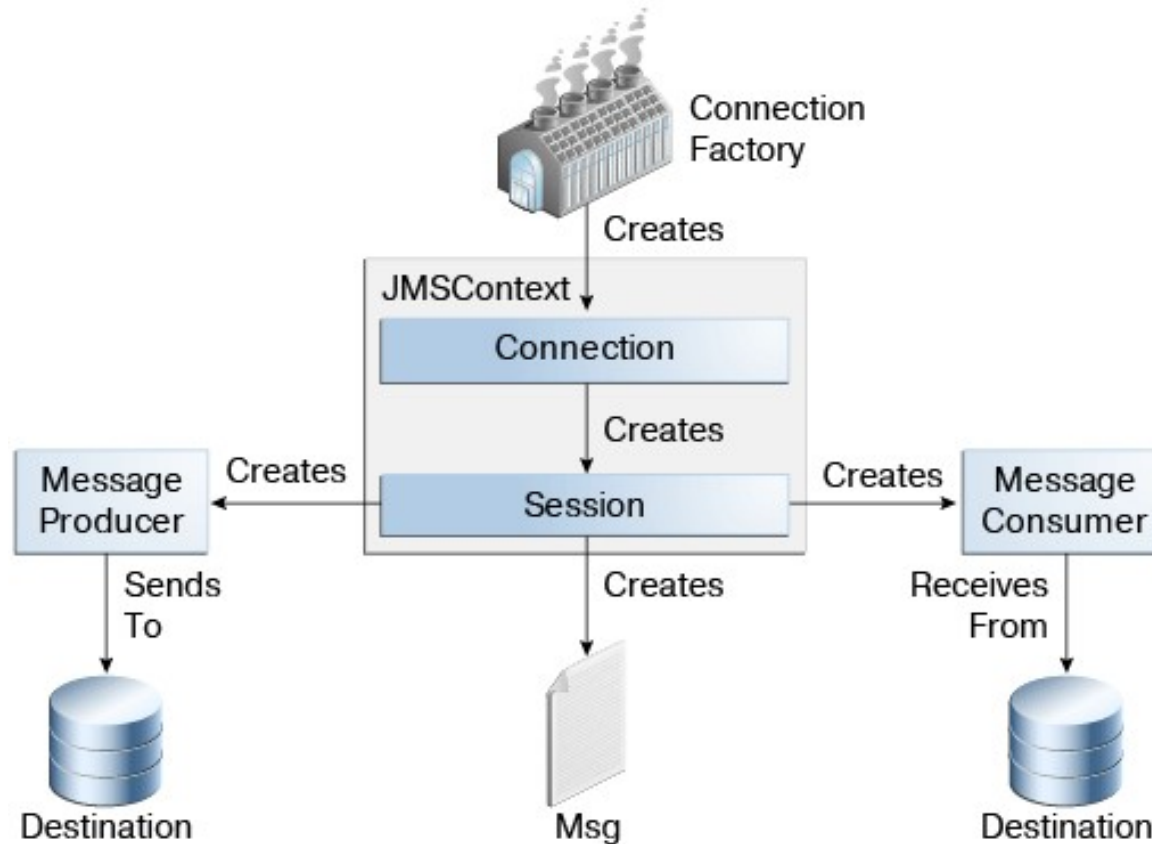
- A **point-to-point** (PTP) product or application is built on the concept of message **queues**, **senders**, and **receivers**.
  - Each message has only one consumer.
  - The receiver can fetch the message whether or not it was running when the client sent the message.



- In a **publish/subscribe** (pub/sub) product or application, clients address messages to a **topic**, which functions somewhat like a bulletin board. Publishers and subscribers can dynamically publish or subscribe to the topic.
  - Each message can have multiple consumers.
  - A client that subscribes to a topic can consume only messages sent *after* the client has created a subscription, and the consumer must continue to be active in order for it to consume messages.



# The JMS API Programming Model





- Get a reference to `ConnectionFactory`
- A **connection factory** is the object a client uses to create a connection to a provider.

```
import javax.naming.*;
import javax.jms.*;
QueueConnectionFactory queueConnectionFactory;
Context messaging = new InitialContext();
queueConnectionFactory = (QueueConnectionFactory)
    messaging.lookup("QueueConnectionFactory");
```

- Or

```
@Resource(lookup = "java:comp/DefaultJMSConnectionFactory")
private static ConnectionFactory connectionFactory;
```

- A **destination** is the object a client uses to specify the target of messages it produces and the source of messages it consumes.
  - In the PTP messaging style, destinations are called queues.
  - In the pub/sub messaging style, destinations are called topics.

```
Queue queue;  
queue = (Queue)messaging.lookup("theQueue");  
Topic topic;  
topic = (Topic)messaging.lookup("theTopic");
```

- Or

```
@Resource(lookup = "jms/MyQueue")  
private static Queue queue;  
@Resource(lookup = "jms/MyTopic")  
private static Topic topic;
```

- Get a Connection and a Session
- A **connection** encapsulates a virtual connection with a JMS provider.
- A **session** is a single-threaded context for producing and consuming messages.
  - You normally create a session (as well as a connection) by creating a JMSContext object

```
JMSContext context = connectionFactory.createContext();
```

- Create a Producer or a Consumer

```
try (JMSContext context = connectionFactory.createContext();) {  
    JMSProducer producer = context.createProducer();  
    ...  
    context.createProducer().send(dest, message);  
}
```

- Or

```
try (JMSContext context = connectionFactory.createContext();) {  
    JMSConsumer consumer = context.createConsumer(dest);  
    ...  
    Message m = consumer.receive();  
    Message m = consumer.receive(1000);  
}
```

- Create a message

```
TextMessage message = context.createTextMessage();  
message.setText(msg_text); // msg_text is a String  
context.createProducer().send(message);
```

```
Message m = consumer.receive();  
if (m instanceof TextMessage) {  
    String message = m.getBody(String.class);  
    System.out.println("Reading message: " + message);  
} else {  
    // Handle error or process another message type  
}
```

- JMS Message Listener
- A message listener is an object that acts as an asynchronous event handler for messages.

```
Listener myListener = new Listener();  
consumer.setMessageListener(myListener);
```

Listener:

```
void onMessage(Message inMessage)
```

- Selection of messages

- Producer:

```
String data;  
TextMessage message;  
message = session.createTextMessage();  
message.setText(data);  
message.setStringProperty("Selector", "Technology");
```

- Selection of messages

- Consumer:

```
String selector;
```

```
selector = new String(" (Selector = 'Technology') ");
```

```
JMSConsumer consumer = context.createConsumer(dest, selector);
```



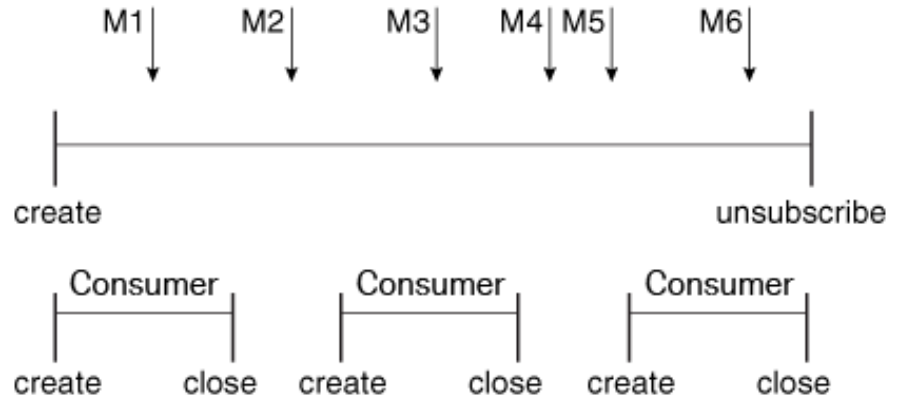
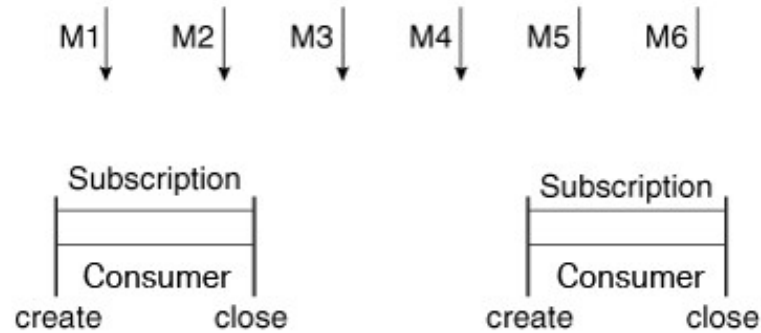
- Durable subscription

```
String subName = "MySub";  
JMSConsumer consumer = context.createDurableConsumer(myTopic,  
subName);
```

```
consumer.close();  
context.unsubscribe(subName);
```

```
JMSConsumer consumer =  
    context.createSharedDurableConsumer(topic, "MakeItLast");
```

# JMS programming model

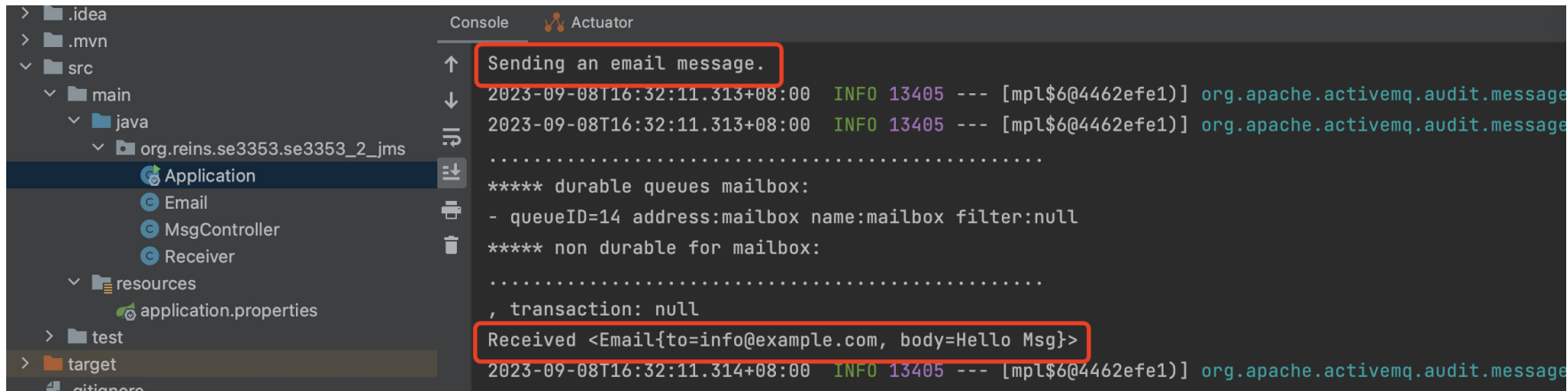


- JMS Browser
  - allows you to browse the messages in the queue and display the header values for each message.

```
QueueBrowser browser = context.createBrowser(queue);
```

- JMS Exception Handling
  - The root class for all checked exceptions in the JMS API is `JMSEException`

- Spring provides the means to publish messages to any POJO (Plain Old Java Object).

The screenshot shows an IDE with a project structure on the left and a console window on the right. The project structure includes a 'src' directory with 'main' and 'test' subdirectories. The 'main' directory contains a 'java' package with 'Application', 'Email', 'MsgController', and 'Receiver' classes, and a 'resources' directory with an 'application.properties' file. The console window shows the following output:

```
Console  Actuator
Sending an email message.
2023-09-08T16:32:11.313+08:00 INFO 13405 --- [mpl$6@4462efe1]] org.apache.activemq.audit.message
2023-09-08T16:32:11.313+08:00 INFO 13405 --- [mpl$6@4462efe1]] org.apache.activemq.audit.message
.....
***** durable queues mailbox:
- queueID=14 address:mailbox name:mailbox filter:null
***** non durable for mailbox:
.....
, transaction: null
Received <Email{to=info@example.com, body=Hello Msg}>
2023-09-08T16:32:11.314+08:00 INFO 13405 --- [mpl$6@4462efe1]] org.apache.activemq.audit.message
```

- Spring provides the means to publish messages to any POJO (Plain Old Java Object).

Email.java

```
public class Email {
```

```
    private String to;  
    private String body;
```

```
    public Email() {  
    }
```

```
    public Email(String to, String body) {  
        this.to = to;  
        this.body = body;  
    }
```

```
    public String getTo() {  
        return to;  
    }
```

```
    public void setTo(String to) {  
        this.to = to;  
    }
```

```
    public String getBody() {  
        return body;  
    }
```

```
    public void setBody(String body) {  
        this.body = body;  
    }
```

```
    @Override  
    public String toString() {  
        return String.format("Email{to=%s, body=%s}", getTo(), getBody());  
    }  
}
```

- This POJO is quite simple, containing two fields (**to** and **body**), along with the presumed set of getters and setters.

- Message Receiver

Receiver.java

```
package org.reins.se3353.se3353_2_jms;
```

```
import org.springframework.jms.annotation.JmsListener;
```

```
import org.springframework.stereotype.Component;
```

```
@Component
```

```
public class Receiver {
```

```
    @JmsListener(destination = "mailbox", containerFactory = "myFactory")
```

```
    public void receiveMessage(Email email) {
```

```
        System.out.println("Received <" + email + ">");
```

```
    }
```

```
}
```

- Send and receive JMS messages with Spring

Application.java

```
package org.reins.demo;
@SpringBootApplication
@EnableJms
public class Application {
    @Bean
    public JmsListenerContainerFactory<?> myFactory(ConnectionFactory connectionFactory,
                                                    DefaultJmsListenerContainerFactoryConfigurer configurer) {
        DefaultJmsListenerContainerFactory factory = new DefaultJmsListenerContainerFactory();
        // This provides all boot's default to this factory, including the message converter
        configurer.configure(factory, connectionFactory);
        // You could still override some of Boot's default if necessary.
        return factory;
    }
```

```
@Bean // Serialize message content to json using TextMessage
public MessageConverter jacksonJmsMessageConverter() {
    MappingJackson2MessageConverter converter = new MappingJackson2MessageConverter();
    converter.setTargetType(MessageType.TEXT);
    converter.setTypeIdPropertyName("_type");
    return converter;
}
```



- Send and receive JMS messages with Spring

`Application.java`

```
public static void main(String[] args) {  
    // Launch the application  
    ConfigurableApplicationContext context = SpringApplication.run(Application.class, args);  
  
    JmsTemplate jmsTemplate = context.getBean(JmsTemplate.class);  
  
    // Send a message with a POJO - the template reuse the message converter  
    System.out.println("Sending an email message.");  
    jmsTemplate.convertAndSend("mailbox", new Email("info@example.com", "Hello"));  
}  
  
}
```

- Send and receive JMS messages with Spring

MsgController.java

@RestController

public class MsgController {

    @Autowired

    WebApplicationContext applicationContext;

    @GetMapping(value = "/msg")

    public void findOne() {

        JmsTemplate jmsTemplate = applicationContext.getBean(JmsTemplate.class);

        // Send a message with a POJO - the template reuse the message converter

        System.out.println("Sending an email message.");

        jmsTemplate.convertAndSend("mailbox", new Email("info@example.com", "Hello Msg"));

    };

}

- Configuration

`application.properties`

`spring.artemis.mode=embedded`

`pom.xml`

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-artemis</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-json</artifactId>
</dependency>

<dependency>
  <groupId>org.apache.activemq</groupId>
  <artifactId>artemis-jakarta-server</artifactId>
  <scope>runtime</scope>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <version>RELEASE</version>
  <scope>compile</scope>
</dependency>
```

- Messaging with JMS
  - <https://spring.io/guides/gs/messaging-jms/>
- Java Message Service Concepts
  - <https://javaee.github.io/tutorial/jms-concepts.html>
- Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example
  - <https://www.confluent.io/resources/ebook/mastering-kafka-streams-and-ksqldb/>



Thank You!