# java message service

marek konieczny

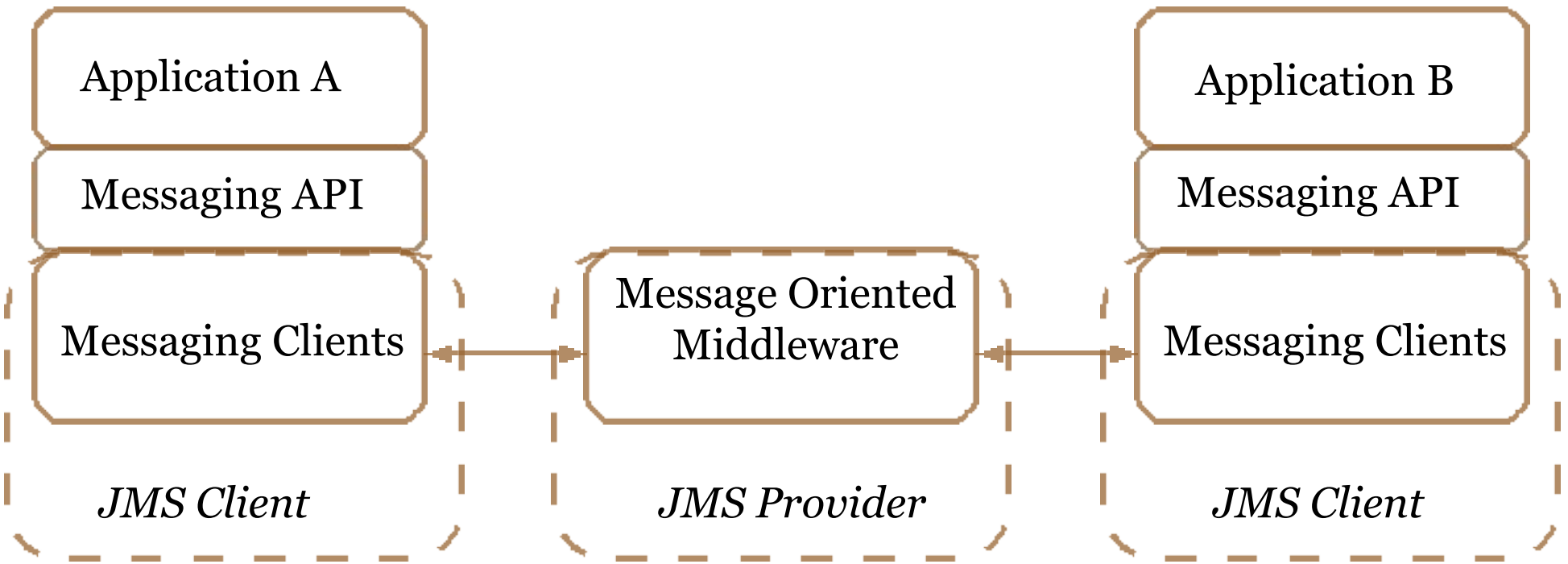
# Agenda

- Introduction to message oriented computing
  - basic communication models and domains
- Java Message Service API
  - Communication API
  - Message structure
  - Selectors API
- Hands-on example, assignment

# Message Oriented Middleware

- Integration issues in  information systems
  - Asynchronous communication.
- Message Oriented Middleware (MOM)
- Why we want to use MOM ?
  - Easy integration of heterogeneous systems,
  - Good solution for the bottlenecks in system design,
  - Overall throughput of the system can increase,
  - Improvement in system architecture flexibility,
  - Allows to build geographically distributed systems.

# Message Oriented Middleware

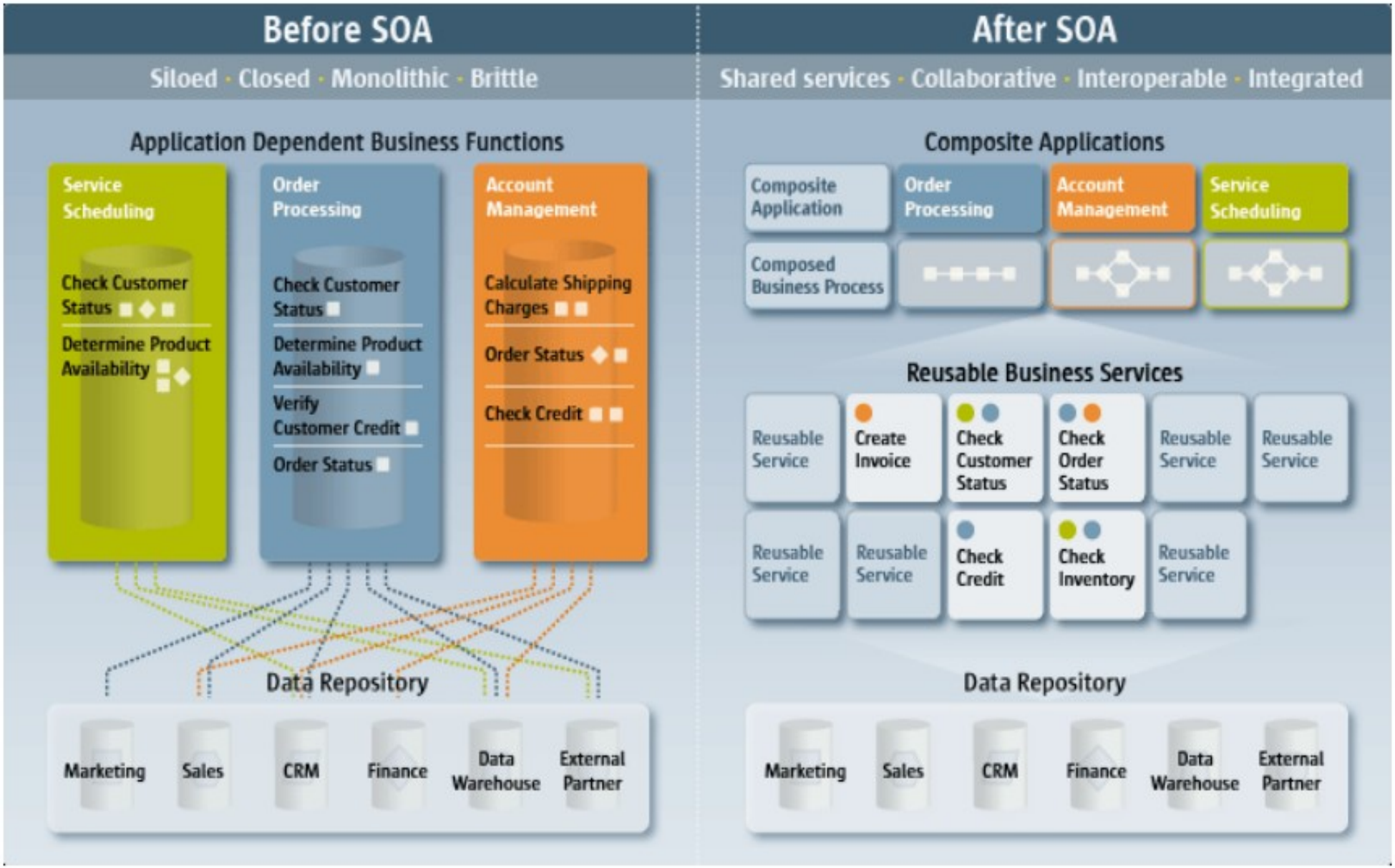


# Service Oriented Architecture

- Enterprise Service Bus (ESB) approach
  - Messages are delivered asynchronously through the network,
  - Application creates a message using simple API and then transport it through the MOM,
  - The messages are autonomous units, they contain all data and states which are required by business logic.

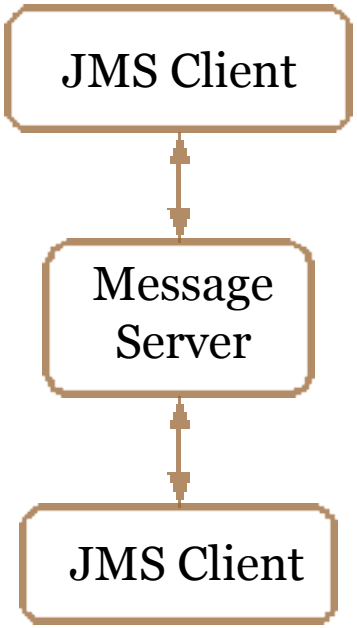
- *Event-driven* approach
  - The communication is done in asynchronous scheme,
  - The messages are sent in efficient and robust way,
  - They are self-described – contain all necessary context that allows to recipients to process it in independent way,
  - All components within the system are loosely-coupled.

# Service Oriented Architecture

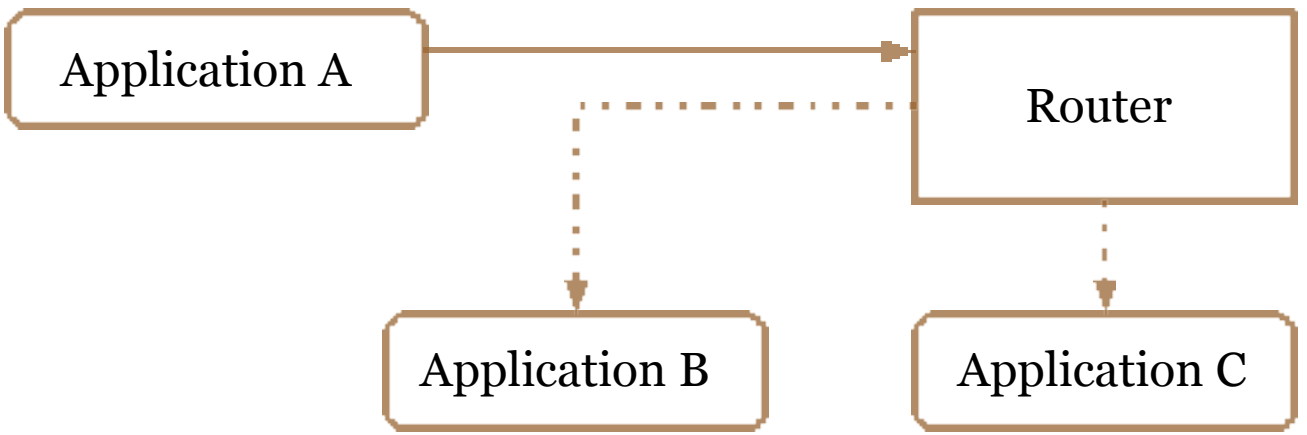


# Architecture for SOA

- *Centralized* architectures
  - A messages server (router or broker) is responsible for delivering messages.




- *Decentralized* architectures
  - Usually use IP multicast at the network level,
  - The server not responsible for routing, it is done on network layer.



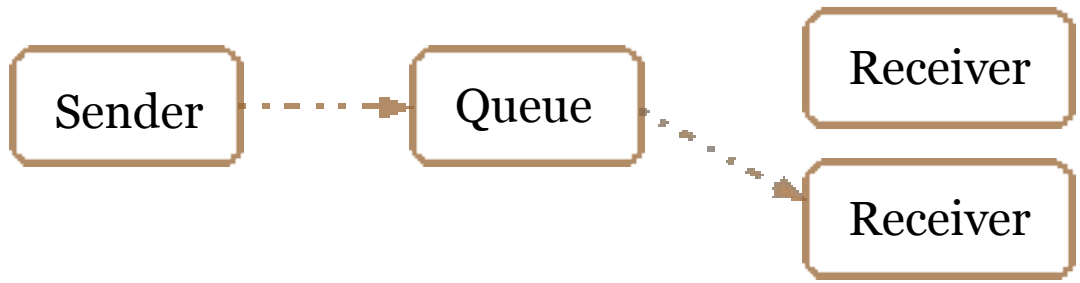
# Communication models for JMS

- *Synchronous* communication
  - Both communication parties need to be active,
  - Sender receives confirmation from receiver,
  - Blocking calls,
  - Scenarios when global authorizations are required (e.g. credit cards authorization systems)

- *Asynchronous* communication
  - Both parties do not need to be active during communication,
  - Confirmations are not required,
  - Non-blocking calls,
  - Useful when massive communication processing is required,
  - Allows for efficient usage of hardware resources,

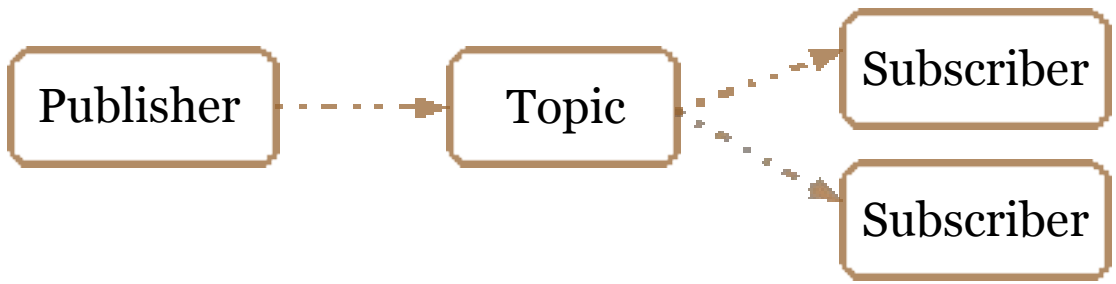
# Point-to-Point domain

- Communication details
  - *Senders* and *receivers* communicate via virtual channels known as *queues* in both asynchronous and synchronous way,
  - Message is received only by one receiver, communication 1-1,
  - Sender can request for new messages at any time,
  - The services are more coupled, the sender usually knows the receiver and is aware of information the receiver is expecting.



# Publish-and-Subscribe domain

- Communication details
  - Messages are published by the virtual channels called *topics,*
  - Producers are called *publishers*, while consumers are called *subscribers,*
  - Messages are *broadcast* to all consumers, every *subscriber* receives a copy of message, communication 1-many
  - The services are less coupled than in *point-to-point* models (publishers do not need to know how many *subscribers* are listening)



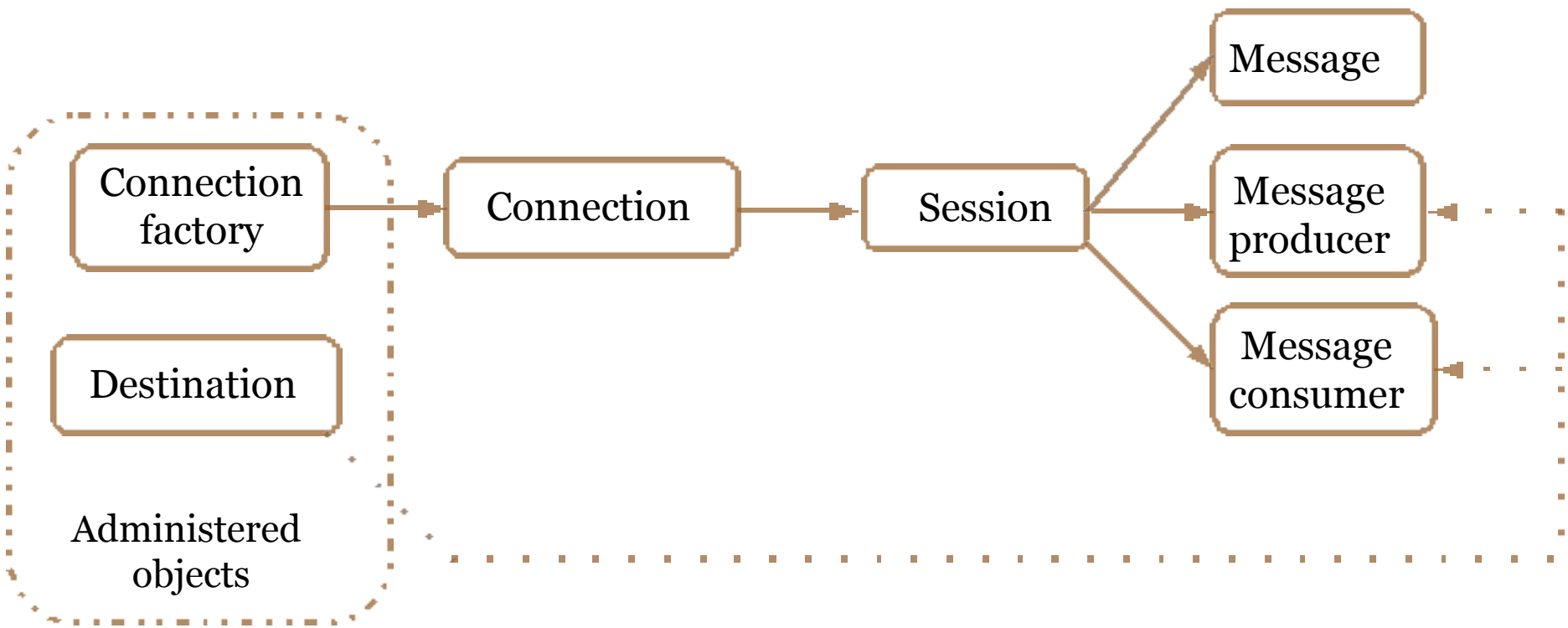
# Java Message Service

- Background
  - JMS is messaging API created by *Sun Microsystem* with cooperation with various MOM vendors,
  - It is just an abstract API not a messaging system – it is only a collection of interfaces and abstract classes,
  - The latest version is JMS 1.1, published in around 2002.

- API can be divided in 3 main parts
  - General API (can be used for interactions with both queues and topics),
  - Point-to-Point API,
  - Publish-and-Subscribe API.

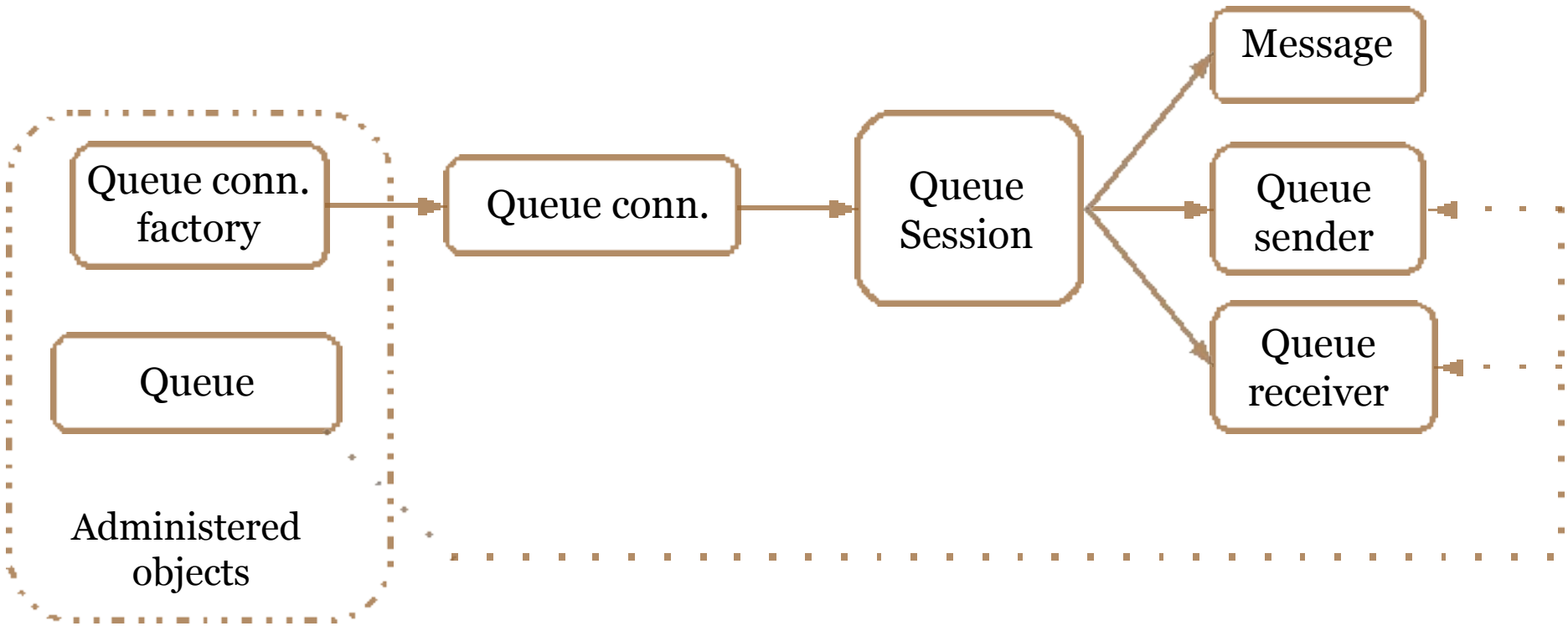
# JMS General API

- Main interfaces
  - *ConnectionFactory, Destination, Connection, Session, Message, MessageProducer, MessageConsumer,*
  - There are other classes for exception handling, message priorities and persistence.



# JMS Point-to-Point API

- Main interfaces
  - *QueueConnectionFactory, Queue, QueueConnection, QueueSession, Message, QueueSender, QueueReceiver,*
  - Most of the interfaces are similar as in the general API – all have *Queue* prefix.

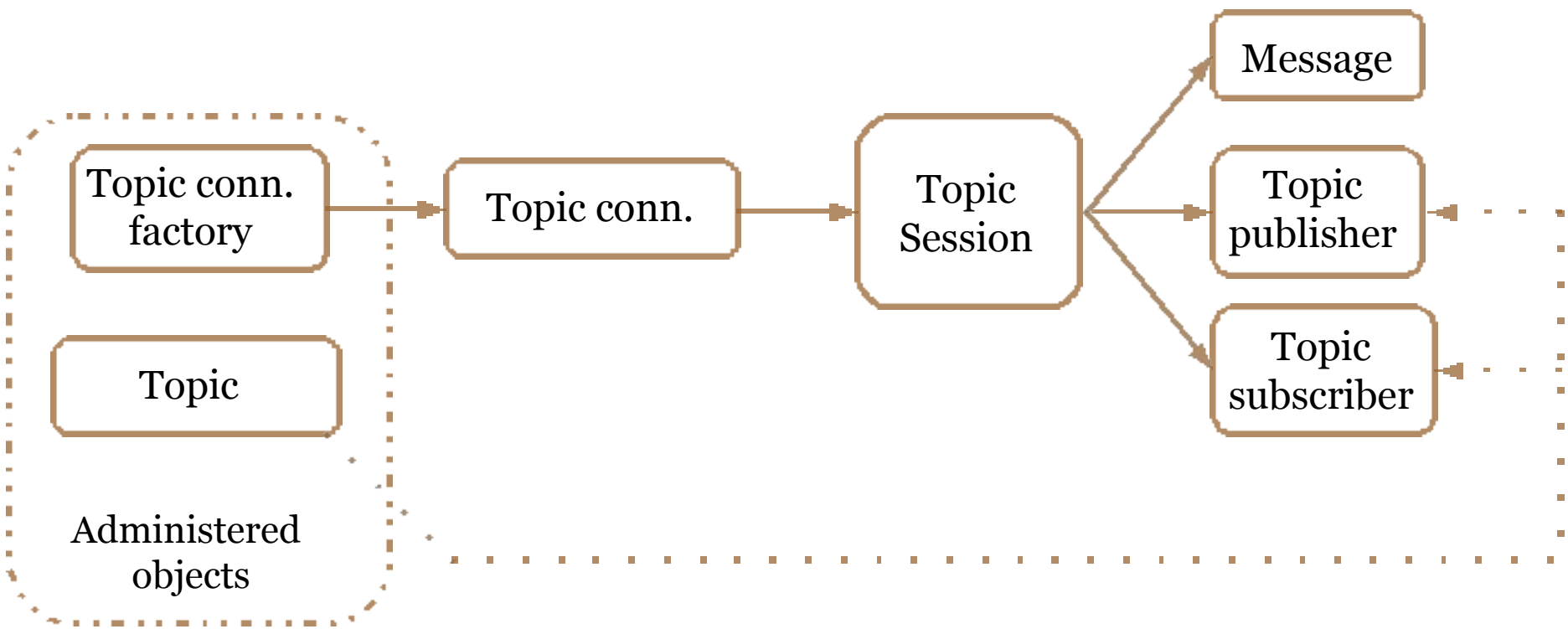


# JMS Point-to-Point (Impl.)

- Producer
  - Obtain reference to *QueueConnectionFactory*,
  - Get reference to *Queue*,
  - Create *QueueConnection*,
  - Create *QueueSession*,
  - Create *QueueSender*,
  - Create *Message*,
  - Send *Message*.
- Consumer
  - Obtain reference to *QueueConnectionFactory*,
  - Get reference to *Queue*,
  - Create *QueueConnection*,
  - Create *QueueSession*,
  - Create *QueueReceiver*,
  - Wait for message, implement interface *MessageListener*.

# JMS Publish-and-Subscribe API

- Main interfaces
  - *TopicConnectionFactory, Topic, TopicConnection, TopicSession, Message, TopicPublisher, TopicSubscriber.*



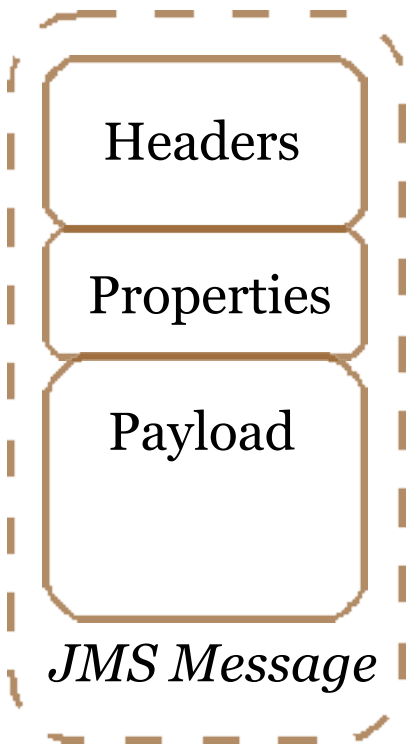
# JMS Publish-and-Subscribe (Impl.)

- Producer
  - Obtain reference to *TopicConnectionFactory*,
  - Get reference to *Topic*,
  - Create *TopicConnection*,
  - Create *TopicSession*,
  - Create *TopicPublisher*,
  - Create *Message*,
  - Send *Message*.
- Consumer
  - Obtain reference to *TopicConnectionFactory*,
  - Get reference to *Topic*,
  - Create *TopicConnection*,
  - Create *TopicSession*,
  - Create *TopicSubscriber*,
  - Wait for message, implement interface *MessageListener*.

# JMS Message API

- The basic and most important class
  - All data and events are transferred by the *Message* objects,
  - The message does not tell receiver what to do.

- It consist of 3 parts
  - The message header,
  - Message properties,
  - Data itself (payload or message body).



# JMS Message Headers

- Basic information:
  - 2 groups, divided by responsible parties: set by developers, set automatically by the java message system,
  - Both can be access by standard *set* and *get* methods.
- Automatic headers
  - *JMSDestination* – defines a destination of a message,
  - *JMSDeliveryMode* – defines persistent or not-persistent delivery mode,
  - *JMSPriority* – set on producer, 0-4 normal and 5-9 expedited.
- Custom headers
  - *JMSReplyTo* – defines a destination of a replay message,
  - *JMSType* – optional header, defines type of a message.

# JMS Message Properties

- Basic information:
  - 3 types : application specific, JMS-defined and provider specific,
  - They function as additional headers to message,
  - The value of property: *String*, *boolean*, *byte*, *double*, *int*, *long*, or *float*.

- Application specific
  - defines any additional data that can be attached to a message.
- JMS-defined properties
  - Automatically set by the JMS provider,
  - *JMSXGroupID, JMSXGroupSeq, JMSXUserID, JMSXAppID* ...
- Provider specific properties
  - Automatically set by the JMS provider,
  - Delivers propriety information of the JMS Provider.

# JMS Message Payload

- Basic information:
  - JMS Provider have to support 6 types of messages: *Message* and *TextMessage, StreamMessage, MapMessage, ObjectMessage, BytesMessage,*
  - Message interface can be extended in order to provide support for other types of messages (e.g. XML).
- Pure *Message* type can be sent
  - if we want to send an event – no payload data
- TextMessage information
  - Carries simple *String* data, standard *get* and *set* method can be use.

```
TextMessage textMessage  = session.createTextMessage();
textMessage.setText("Hello!");
topicPublisher.publish(textMessage)
```

# JMS Message Selectors

- Basic information:
  - Message filtering allows to limit narrow the messages distribution,
  - Instead of filtering everything on the client side we can perform selection on producer site.

```
...
topic = (Topic)ctx.lookup(topicName);
...
String filter = "your condition";
TopicSubscriber subscriber =
    session.createSubscriber(topic, filter, true);
...
```

- Filtering in point-to-point domain:
  - Message filtering is interesting on *queues*, once message is filtered it is removed and not available to others,
  - Here we can use priorities, the rules are first applied to messages with higher priority.

# JMS Message Selectors

- Selectors can be applied to message consumers:
  - *QueueReceiver, QueueBrowser, or TopicSubscriber,*
  - Message headers and properties can be used as data in constructing filters,
  - There is no access to message body.

- Constructing selectors
  - in order to construct rule we need to use SQL-92 conditional expression syntax,
  - We use identifiers for comparison – they come from properties and headers (e.g. *Name = 'abc' AND JMSPriority > 2*),
  - Literals are hard-coded to filter and compared to identifiers,
  - Comparison operators compare them, they produce Boolean value *true* or *false*. They include: *algebraic comparator, and operators LIKE, BETWEEN, IN, NOT* and *IS NULL*.

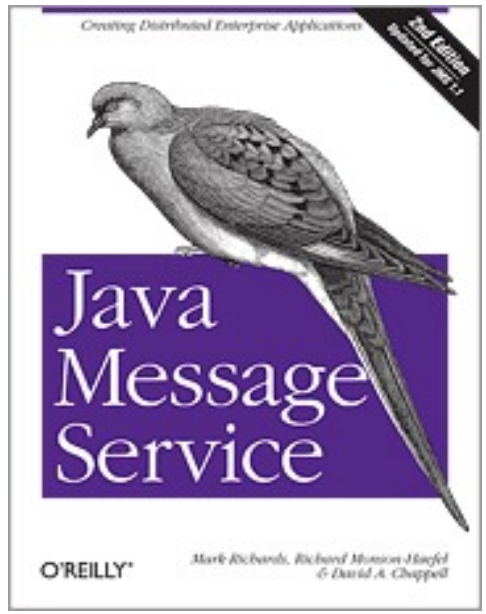
# Assignment

- Functional requirements:
  - Create basic stock quotes broker,
  - Stocks are grouped by the indexes, we have index1 (comp1, comp2, comp3) and index2 (comp4, comp5),
  - Clients can obtain updates of entire index or single stocks,
  - System can update values of single stocks.
- Non-functional requirements:
  - Think about durable subscriptions and security,
  - Implementation should be done using Fuse Message Broker,
  - Applications should use Maven and Spring as much as possible.
- Additional information:
  - Please send your assignment in advance, use prefix *[jms] name surname,*
  - You can use your own HW, expect questions regarding your impl.

# References

- Presentation based on the following materials:
  - Course materials from Network Services Implementation (CS),
  - *Java Message Service 2nd Edition*, By Mark Richards, Richard Monson-Haefel, David A Chappell, Publisher:O'Reilly Media,Released: May 2009.

- Additional materials:
  - Fuse OpenSource website: *http://fusesource.com/*
  - Please review and read the Message Broker docs.



# Demo session

# Questions?