

SPRING DISTRIBUTED SERVICES

***Knowledge is present
everywhere***

Remote Service Definition

- Remote services are services hosted on remote servers
- Accessed by clients over the network.
- Spring features integration classes for remoting support using various technologies.
- The remoting support eases the development of remote-enabled services, implemented by your usual (Spring) POJOs.
- Spring Remoting exposes services over the web for Spring clients to consume as easily as though they were locally instantiated.

Remoting Technologies

RMI- Remote Method Invocation - Use RMI to invoke a remote method. The java objects are serialized. RMI also has firewall issues.

Hessian- Transfer binary data between the client and the server.

Burlap- Transfer XML data between the client and the server. It is the XML alternative to Hessian.

Hessian and Burlap are portable; integrate with other languages such as C# and PHP

Spring's HTTP invoker- Spring provides a special remoting strategy which allows for Java serialization via HTTP, supporting any Java interface .

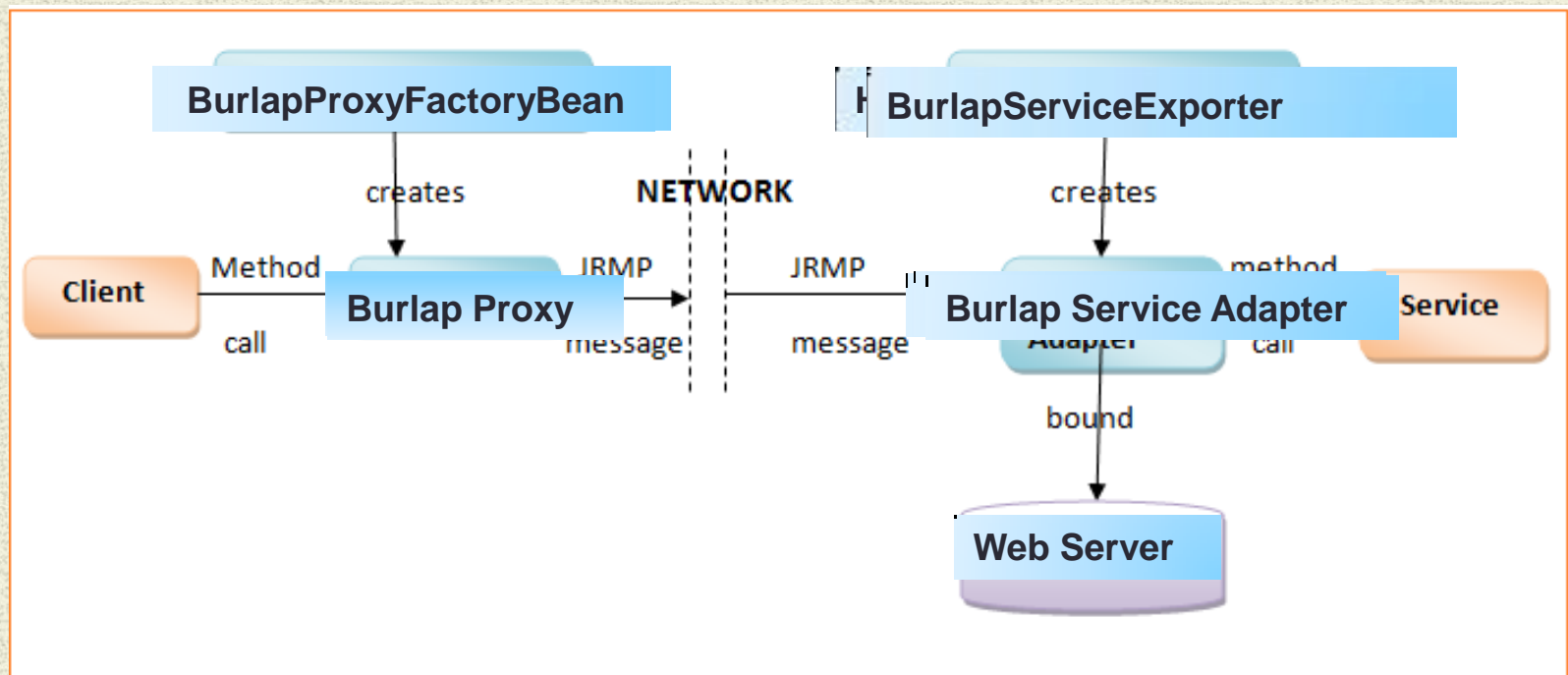
Both client and server applications have to be based on Spring.

JMS- Remoting using JMS is supported via the JmsInvokerServiceExporter and JmsInvokerProxyFactoryBean classes.

AMQP- Remoting using AMQP as underlying protocol - supported by the Spring AMQP project.

JAX-WS. Spring provides remoting support for web services via JAX-WS [SOAP]

RMI, Http, Hessian & Burlap



Configure HTTP Server/Client

SERVER

• `@Bean(name = "/weather")`

```
• public HttpInvokerServiceExporter exporter() {
•   HttpInvokerServiceExporter exporter=new HttpInvokerServiceExporter();
•       exporter.setService(weatherService());
•       exporter.setServiceInterface(WeatherService.class);
•       return exporter;
•   }
```

WeatherService...proxy interface



CLIENT

• `@Bean`

```
• public HttpInvokerProxyFactoryBean weatherService() {
•   HttpInvokerProxyFactoryBean factory =
•       new HttpInvokerProxyFactoryBean();
•   factory.setServiceUrl("http://localhost:8080/HTTPInvokerServer/weather");
•   factory.setServiceInterface(WeatherService.class);
•   return factory;
• }
```


Configure Hessian Server/Client

• SERVER

```
• @Bean(name = "/weather")
• public HessianServiceExporter exporter() {
•     HessianServiceExporter exporter = new
HessianServiceExporter();
•     exporter.setService(weatherService());
•     exporter.setServiceInterface(WeatherService.class);
•     return exporter;
• }
```

WeatherService...proxy interface



• CLIENT

```
• @Bean
• public HessianProxyFactoryBean weatherService() {
•     HessianProxyFactoryBean factory = new HessianProxyFactoryBean();
•     factory.setServiceUrl("http://localhost:8080/HessianServer/weather");
•     factory.setServiceInterface(WeatherService.class);
•     return factory;
• }
```


Main Point

- Distribution of work makes possible better organized more specialized and scalable systems.
- ***Science of Consciousness: Pure Consciousness has infinite organizing power.***

Service Oriented Architecture(SOA)

Some Definitions

Web Service

- Usually single function, e.g., producing data, validating a customer, or providing simple analytical services.

Service-Oriented Architecture

A collection of services.

Sequenced data passing

Multiple services coordinating some activity.

A way of aggregating services to business processes.

Defined by service directories, service governance, Security, SLA, orchestration...

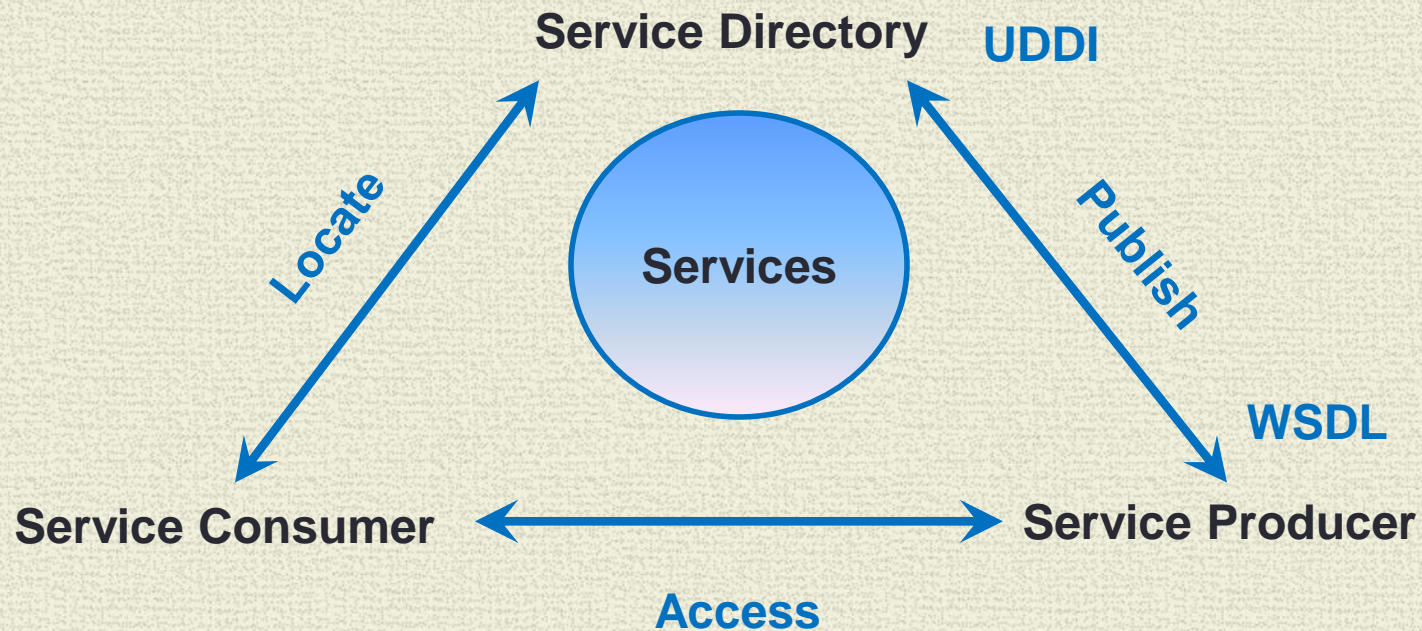
Orchestration

Provides central control over a business process[AKA Workflow].

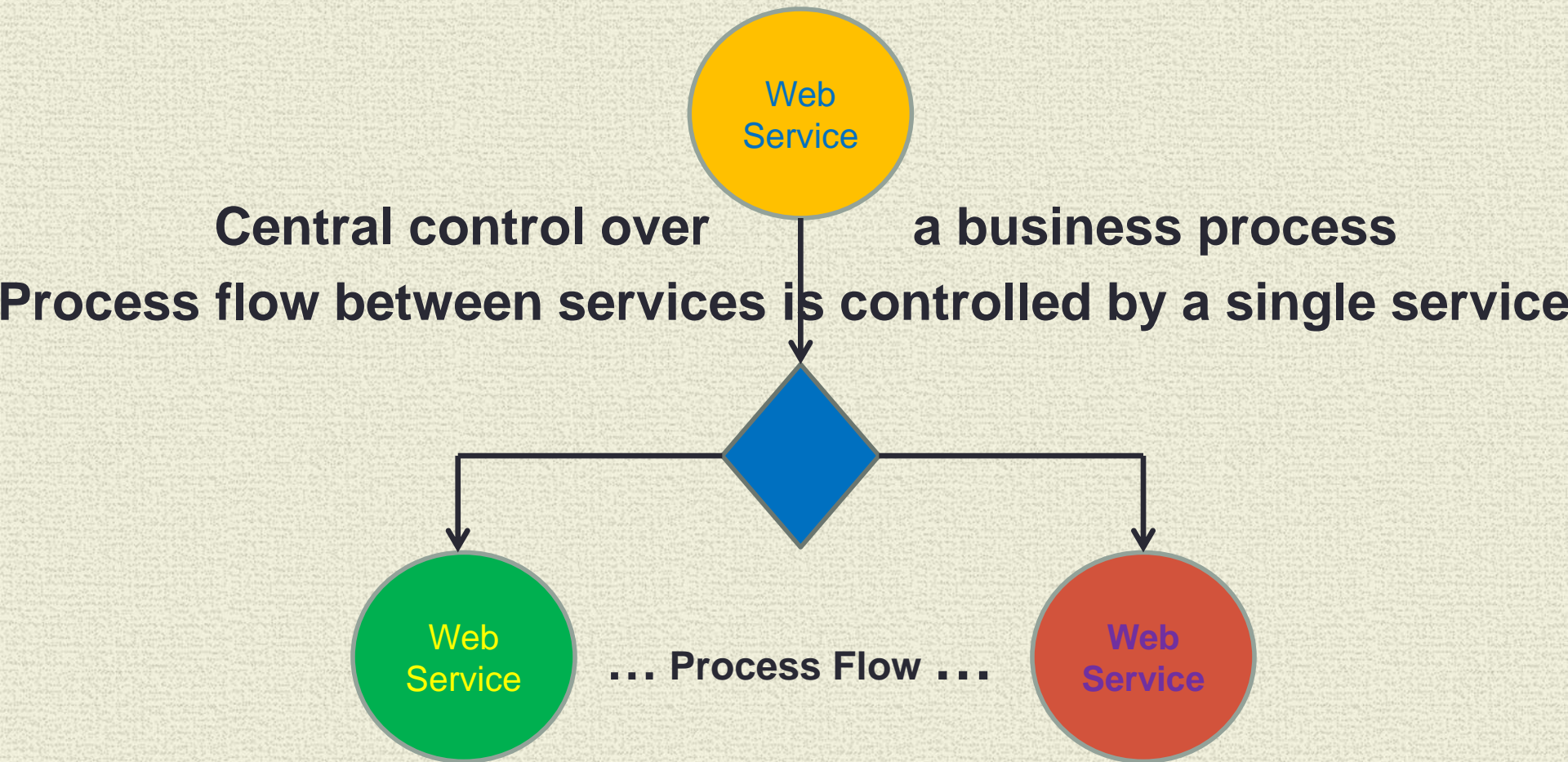
Choreography

Global description of the participating services, a decentralized approach for service composition.

Services Oriented Architecture Publish & Lookup

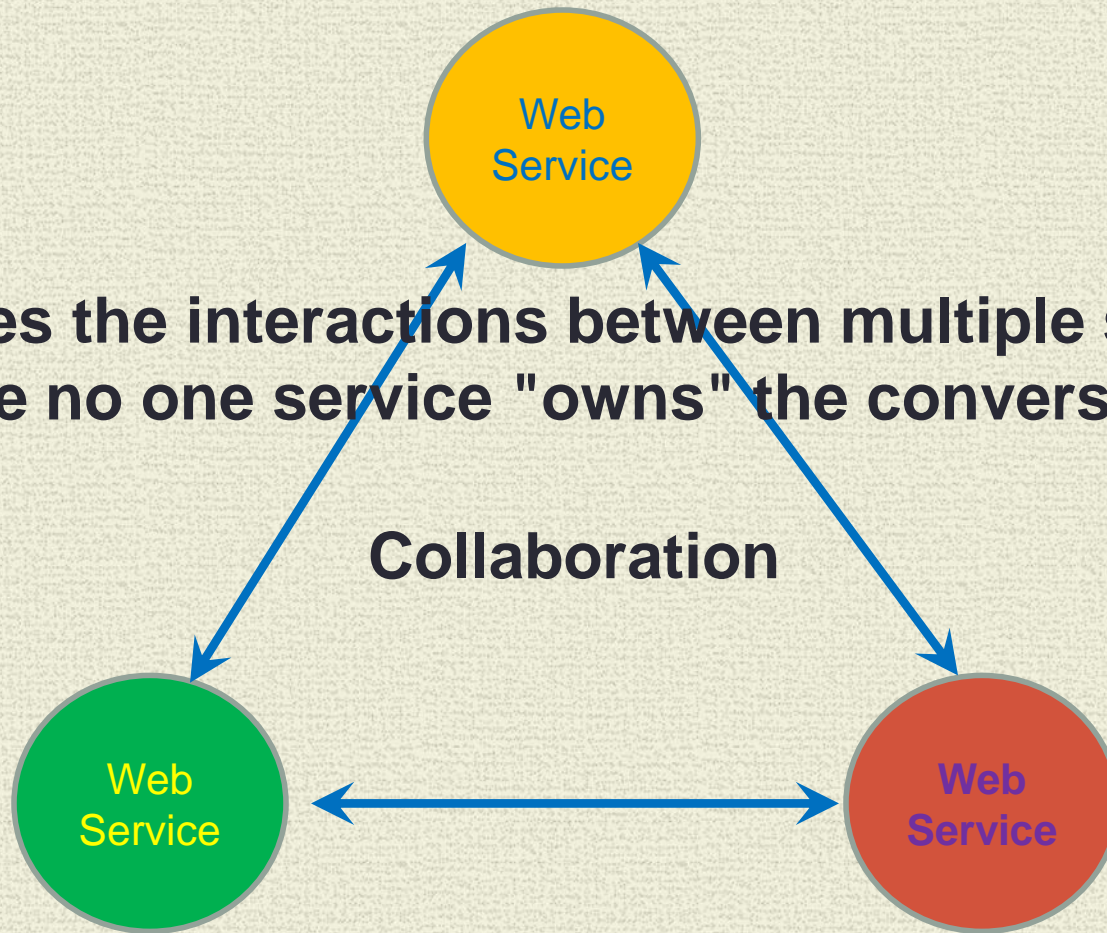


SOA Orchestration



SOA Choreography

Describes the interactions between multiple services, where no one service "owns" the conversation.



Simple Object Access Protocol SOAP

- Language, platform, and transport independent
(not only HTTP)
- Works well in distributed enterprise environments
- Standardized
- Built-in error handling
- Provided document style to better represent domain model
- Strong enterprise adoption – B2B
- **Built-in Security**
- Business Process Language {BPEL}

Web Services Description Language for SOAP

- XML format for describing network services as a set of endpoints operating on messages containing either **document-oriented** or **procedure-oriented** information.

Procedure-oriented : RPC Style

- Defines that the SOAP message body is viewed as a single structure consisting of a method name and a set of parameters.

Document-oriented: Document Style

- Defines of data/information [Domain Object] payload .vs. RPC.
- The operations and messages are described abstractly, and then **bound to a concrete network protocol** and message format to define an endpoint

Web Service Generation

- **Contract-first web service**
- The "contract" (a WSDL definition[**XML**] of operations and endpoints and **XML** schema of the messages) is created first, without actually writing any service code.
- **Contract-last web service**
- Existing logic is "exposed" as a web service and the contract is Generated from it.
- **Spring WS**
 - "Features ...a solution for contract-first, document-driven web services
- highly recommended for building modern, future-proof web services."*

Generate WSDL from an XML schema definition [Domain Model]

Generate Java classes from the WSDL XML schema definition

Artifacts to “Manually” Construct

- **Create the SOAP XML Schema definitions:**
 - AccountDetails.xsd ----- Domain Model Schema
 - AccountDetailsServiceOperations.xsd ----- Request/response schema
- Build using eclipse → run as → Maven Build [goal: clean install]
- **Create SOAP Implementation Classes**
 - AccountService.java ----- Service Interface
 - AccountServiceImpl.java ----- @Service access “dao”
 - AccountServiceEndPoint.java ----- Handles Request
- Eclipse run as → Run on Server
 - Click on URL on displayed Web Page : [Generate WSDL]

Domain Model Schema

```
• <xs:element name="Account" type="Account"/>
  <xs:complexType name="Account">
    <xs:sequence>
      <xs:element name="AccountNumber" type="xs:string"/>
      <xs:element name="AccountName" type="xs:string"/>
      <xs:element name="AccountBalance" type="xs:double"/>
      <xs:element name="AccountStatus" type="EnumAccountStatus"/>
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="EnumAccountStatus">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Active"/>
      <xs:enumeration value="Inactive"/>
    </xs:restriction>
  </xs:simpleType>
```


Generate Domain Classes

Based on XML schema

The right approach is do this automatically during build time using a maven plugin

- `<plugin>`

- `<groupId>org.codehaus.mojo</groupId>`

- `<artifactId>jaxb2-maven-plugin</artifactId>`

- ...

- `<goals>`

- `<goal>xjc</goal>`

- `</goals>`

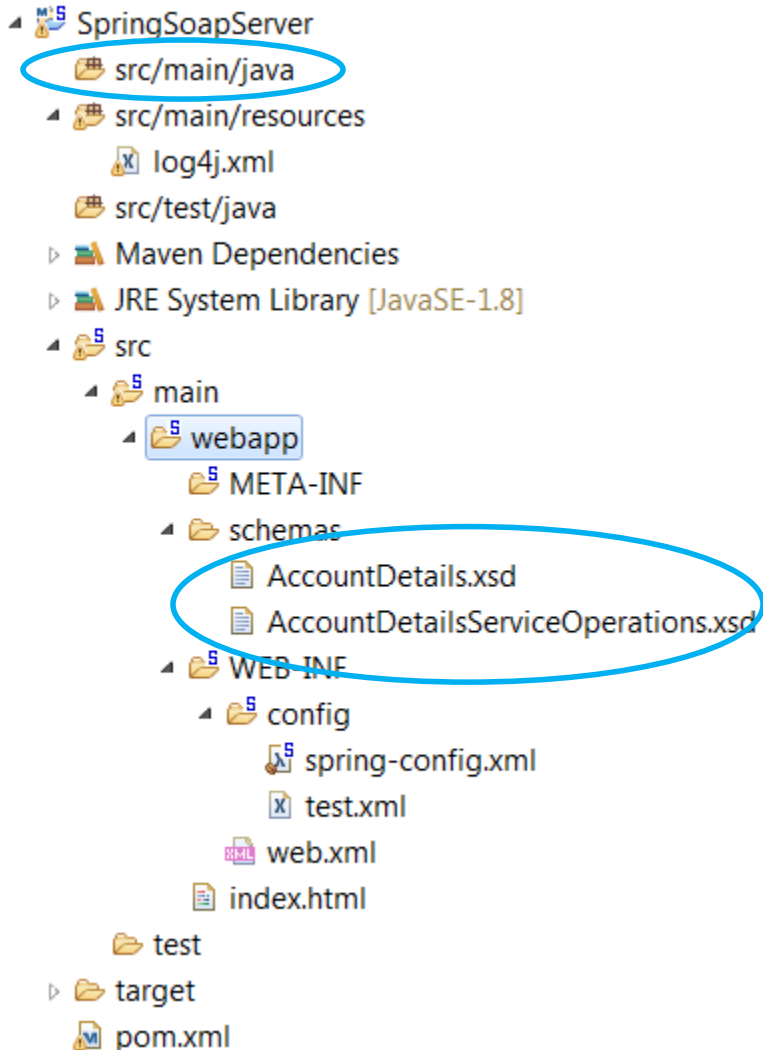
- ...

- `</plugin>`

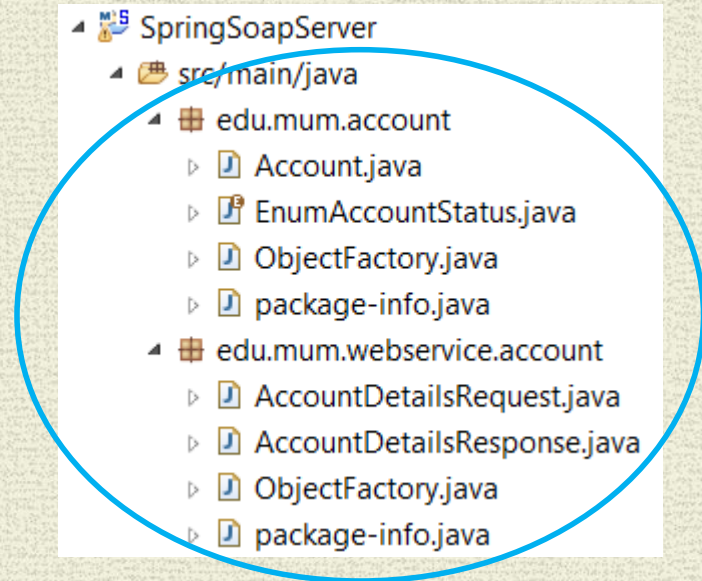
- JAXB simplifies access to an XML document from a Java program - binds the schema for XML document to set of Java classes that represents the schema.
- XJC binding compiler from the JAXB distribution

Soap Server Project

• Before Maven Build

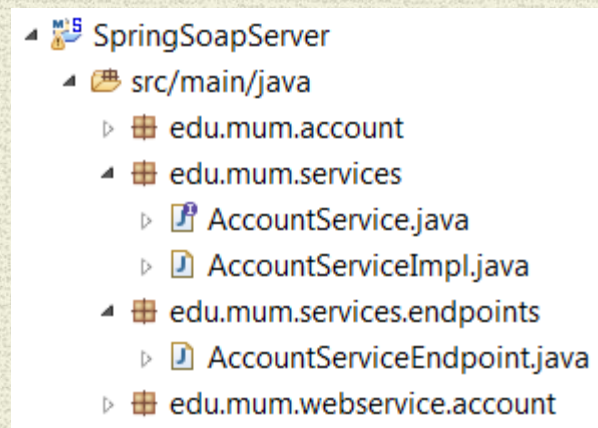


After Maven Build



Create SOAP Implementation Classes

- Server side Implemented Classes to utilize SOAP Service



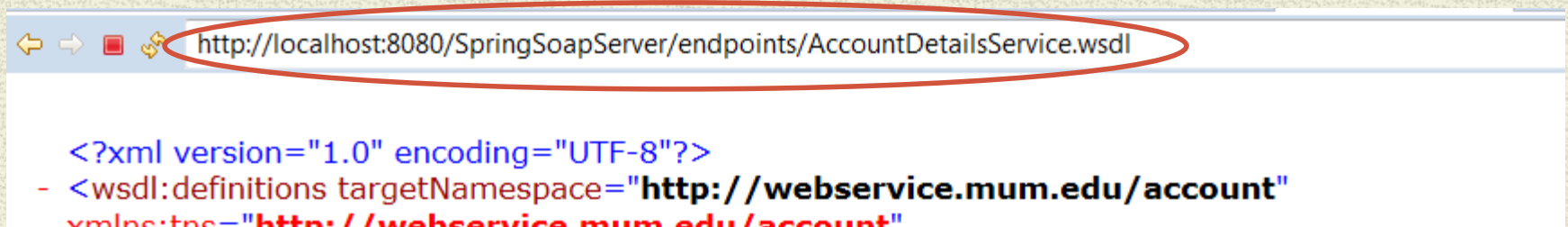
- AccountService; AccountServiceImpl:
Reflect Spring @Service

AccountServiceEndpoint:

Processes SOAP Request/Response Operations/Payloads

SOAP Server - WSDL

- Run As -> Run on Server



SOAP Client

- Generate Java classes from the WSDL XML schema definition

The right approach is do this automatically during build time using a maven plugin

Import WSDL :

eclipse -> import->web service->Launch Web Services Explorer

[upper right] select WSDL page



Click WSDL main – right panel enter WSDL URL

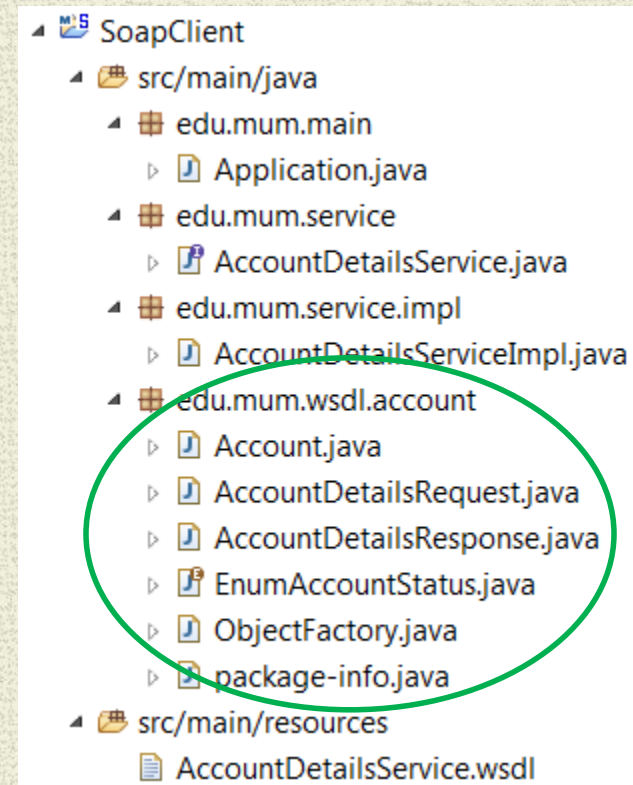
[upper right] select download



- Target: Workbench project : SoapClient

Client Classes Generated from WSDL

- Auto-generated by *maven clean install* based on WSDL
- AccountDetailsService.java
- AccountDetailsServiceImpl.java
- Spring @Service...
- Communicates with SOAP Server via
- Spring WebServiceTemplate



Main Point

- SOAP is characterized by standards and tools based on those standards that automatically generate Client-Server connectivity.
- *The Laws of Nature are conventions [or standards] that spontaneous supports the underlying structure that provides continuity to Life.*

Messaging Systems

[JMS & AMQP]

Loosely coupled - asynchronous - reliable –
communication between applications

Performance

improved response times by doing some tasks asynchronously

Decoupling

Reduced complexity by decoupling and isolating applications

Scalability

Scale distribute tasks across machines based on load

High-quality, cost-effective

Build apps based on specific function - easier to develop, debug, test

High availability

Robustness and reliability- message queue persistence -

- potential zero-downtime redeploys

JMS & AMQP

JMS has queues and topics.

- A message sent on a queue is consumed by no more than one client.
- A message sent on a topic may be consumed by multiple consumers.

AMQP only has queues....and exchanges

- Queues are only consumed by a single receiver
- AMQP doesn't publish directly to queues.

A message is published to an exchange
routed to one queue or multiple queues
Emulating [JMS] queues and topics.

Java Messaging Service

- ◆ A **specification**[JSR 914] that describes a common way for **Java programs** to create, send, receive and read distributed enterprise messages
- ◆ *loosely coupled* communication
- ◆ *Asynchronous* messaging
- ◆ *Reliable* delivery
 - A message is guaranteed to be delivered once and only once.
- ◆ Outside the specification
 - **Security services**
 - **Management services**

Terminology

Broker

Responsible for receiving, routing, and dispensing messages to consumers.

- **Client**

Application - uses message **broker** to communicate with other applications.

- **Consumer**

Application that consumes messages from a messaging **destination**.

- **Destination**

Holding area for messages in **broker**. Clients publish/consume from...

- **Durable Subscriber**

Consumer receives all messages published on a topic- even after inactive

- **Message**

An atomic unit of data that is passed between two or more clients.

- **Producer**

Application that creates and posts messages to a messaging **destination**.

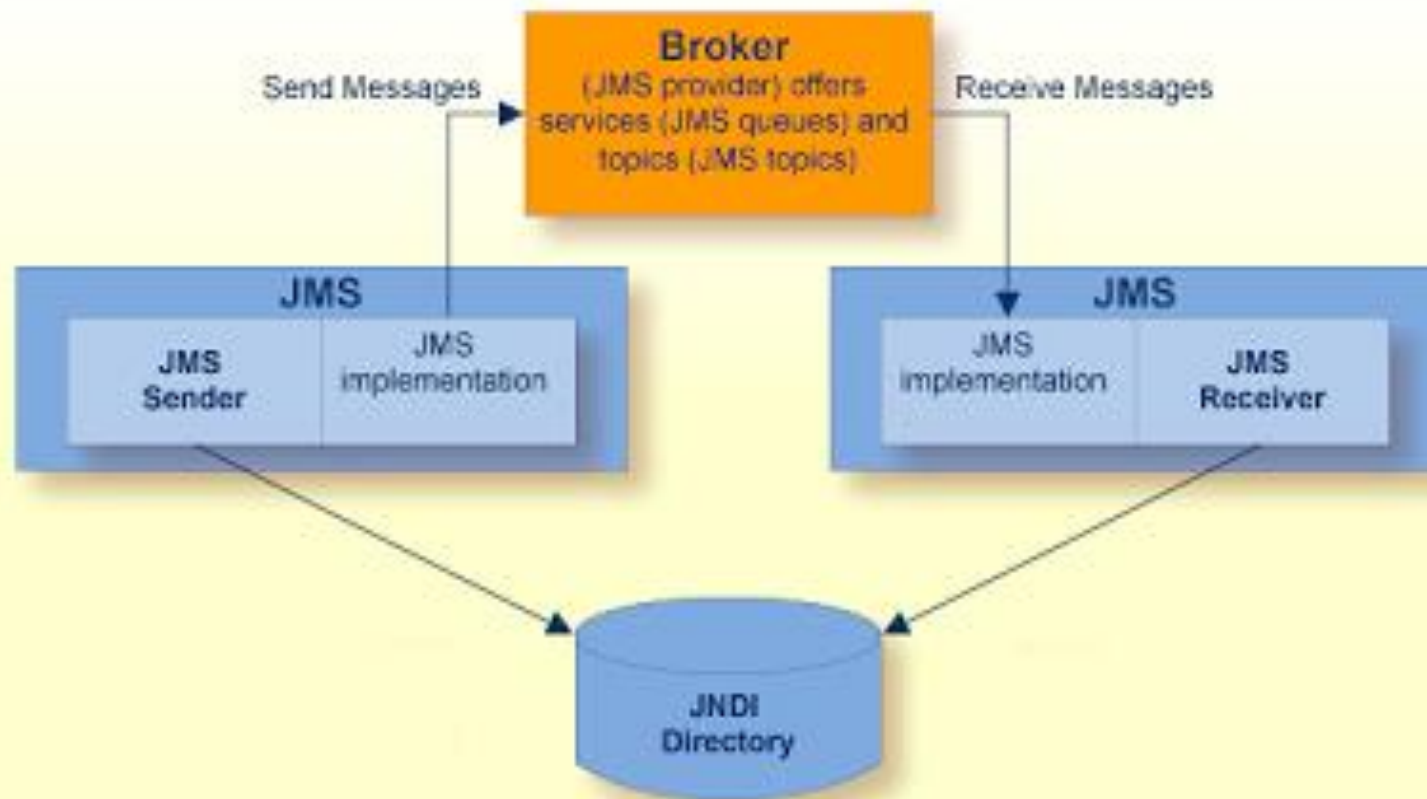
- **Queue**

A **destination** that uses first in/first out semantics.

- **Topic**

A **destination** that uses publish and subscribe semantics.

Decoupled Distributed Messaging [JMS Example]



Main Point

Messaging oriented services guarantee a reliable communication and simplify the complexity of the applications.

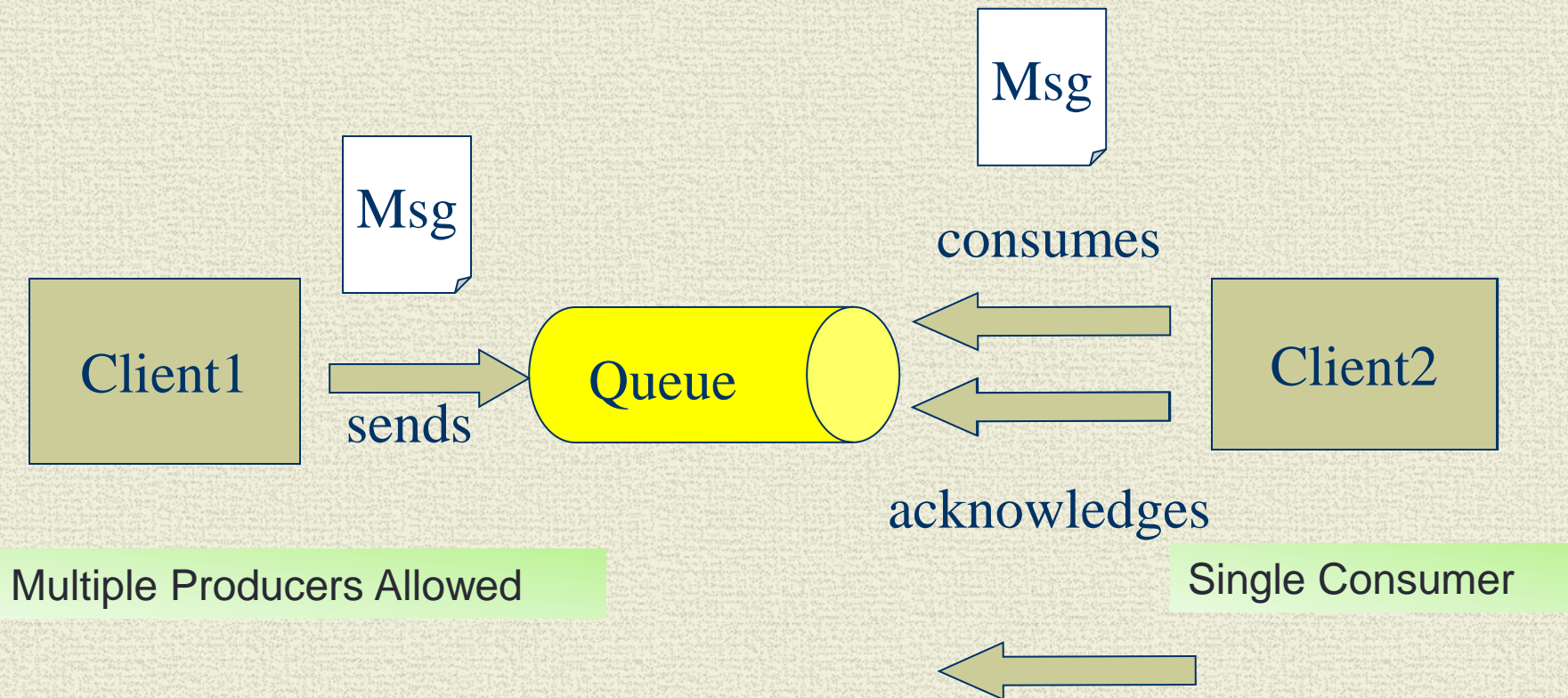
Science of Consciousness: Pure Consciousness is simple, reliable, efficient and precise.

JMS Services

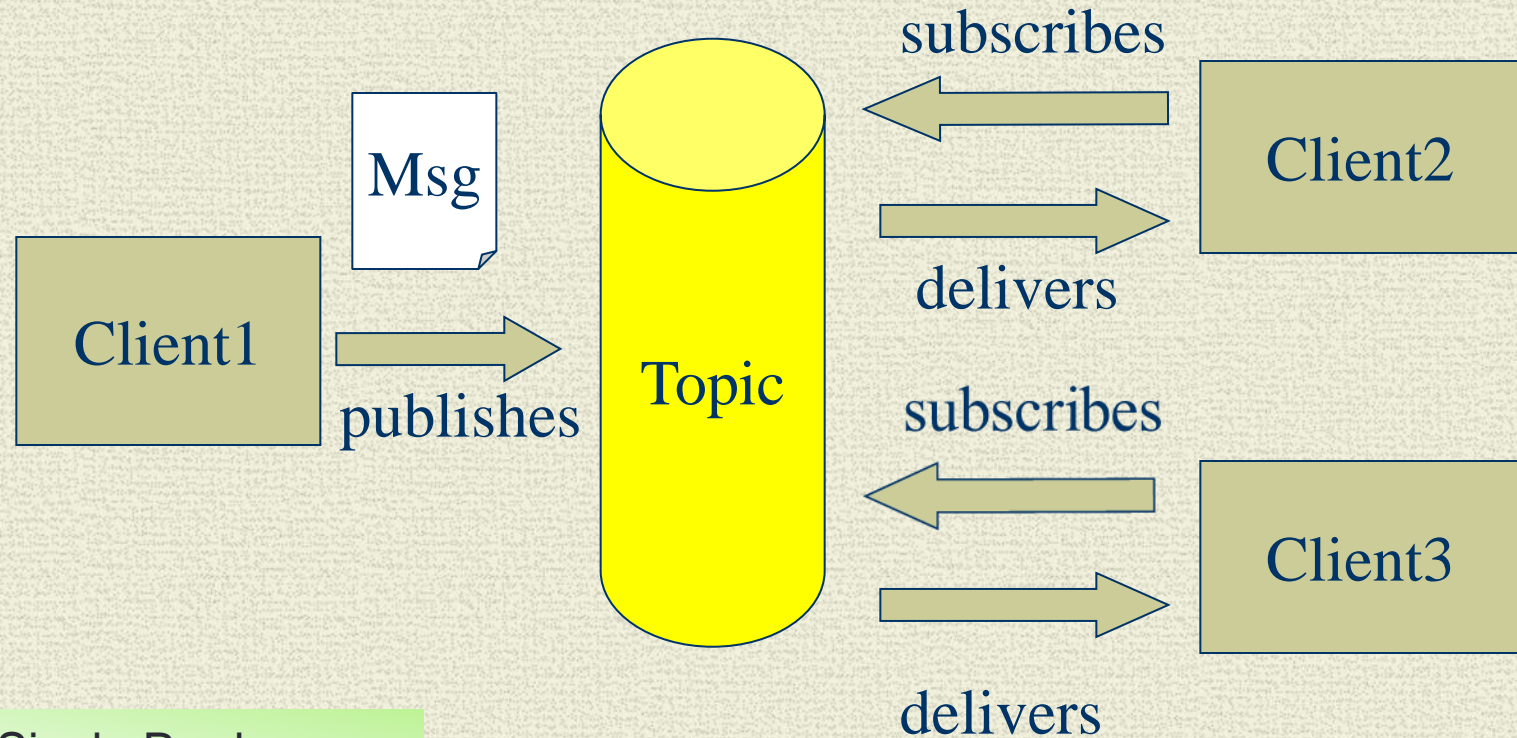
- ◆ Point-to-Point (PTP)
 - Built around the concept of a message queue
 - Each message has only one consumer
 - Multiple producers

- ◆ Publish-Subscribe systems
 - Uses a “topic” to send and receive messages
 - Each message has multiple consumers
 - Single producer

Point-to-Point



Publish/Subscribe



Single Producer

Multiple Consumers Allowed

◆ Synchronously

- A subscriber or a receiver explicitly fetches the message from the destination by calling the receive method.
- The receive method can *block* until a message arrives or can time out if a message does not arrive within a specified time limit.

◆ Asynchronously

- A client can register a *message listener* with a consumer.
- Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's `onMessage()` method.

Message Types

- **TextMessage** A java.lang.String object
- **MapMessage** A set of name/value pairs
- **BytesMessage** A stream of uninterpreted bytes
-
- **StreamMessage** A stream of primitive values
- ***ObjectMessage*** A Serializable object

Demo

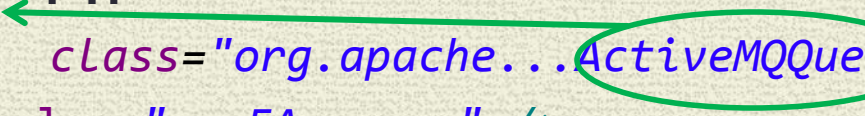
- **Connect to BROKER**

```
<bean id="connFactory" class="org.apache...ActiveMQConnectionFactory"
  <p:brokerURL="tcp://localhost:61616"

```


- **Assign Broker Queue -- PTP**

```
<bean id="mumEAQueue" class="org.apache...ActiveMQQueue">
  <constructor-arg value="mumEA.queue" />
```



- **Create Queue Producer**

```
<bean id="jmsTemplate" class="org...jms.core.JmsTemplate">
  <p:name="defaultDestinationName" value="mumEAQueue"/>
```

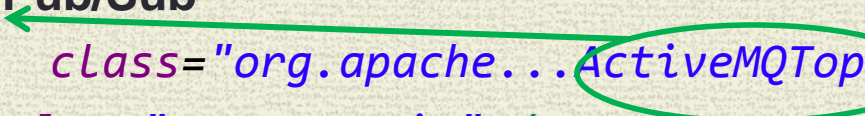


- **Create Queue Consumer**

```
<jms:listener destination="mumEAQueue" ref="ptpMessageListener"
```

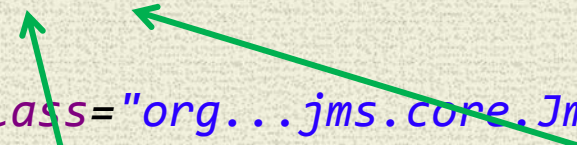
- **Assign Broker Topic – Pub/Sub**

```
<bean id="mumEATopic" class="org.apache...ActiveMQTopic">
  <constructor-arg value="mumEA.topic" />
```



- **Create Topic Publisher**

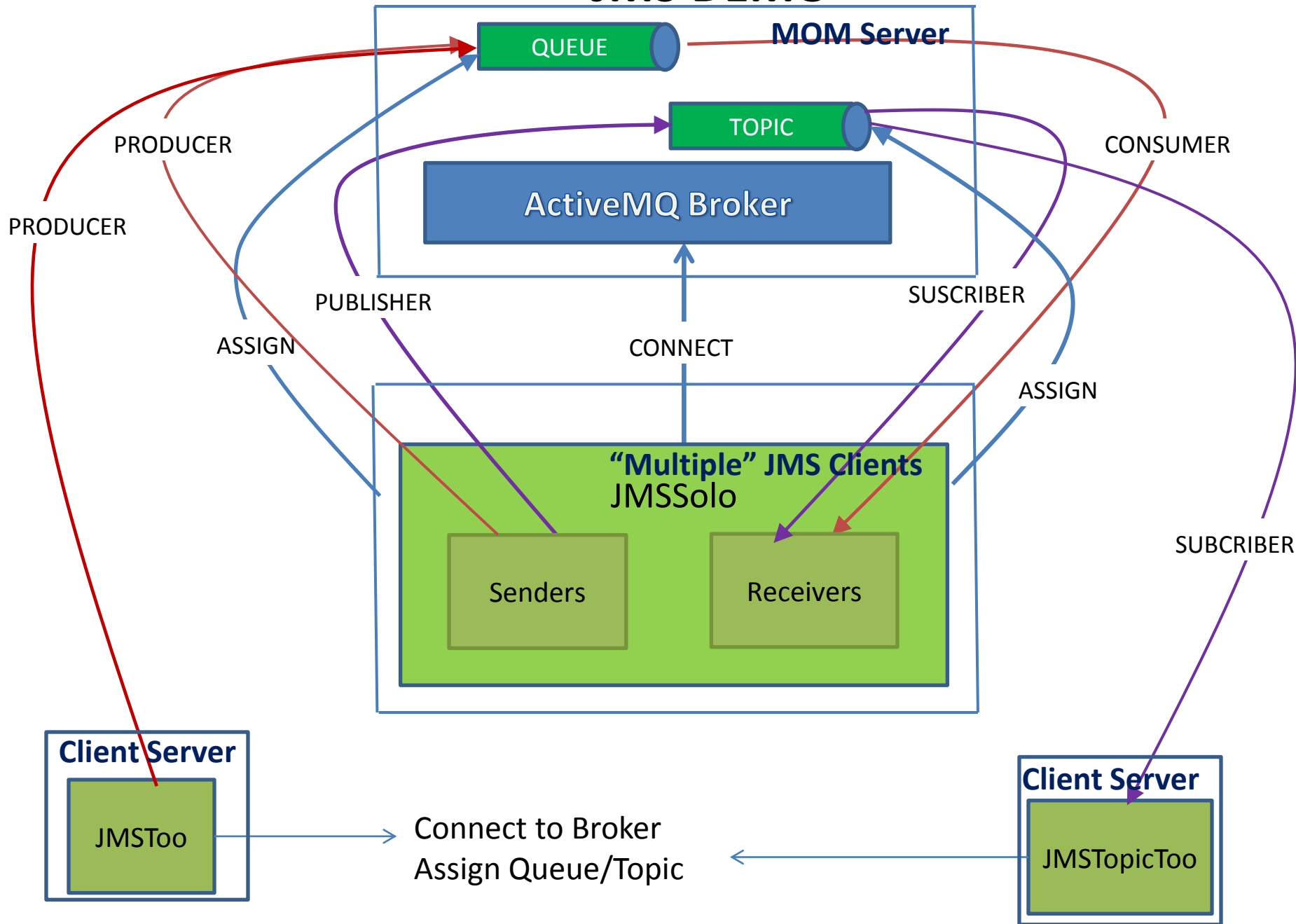
```
<bean id="jmsTopicTemplate" class="org...jms.core.JmsTemplate">
  <p:name="defaultDestinationName" value="mumEATopic"/>
```



- **Create Topic Subscriber**

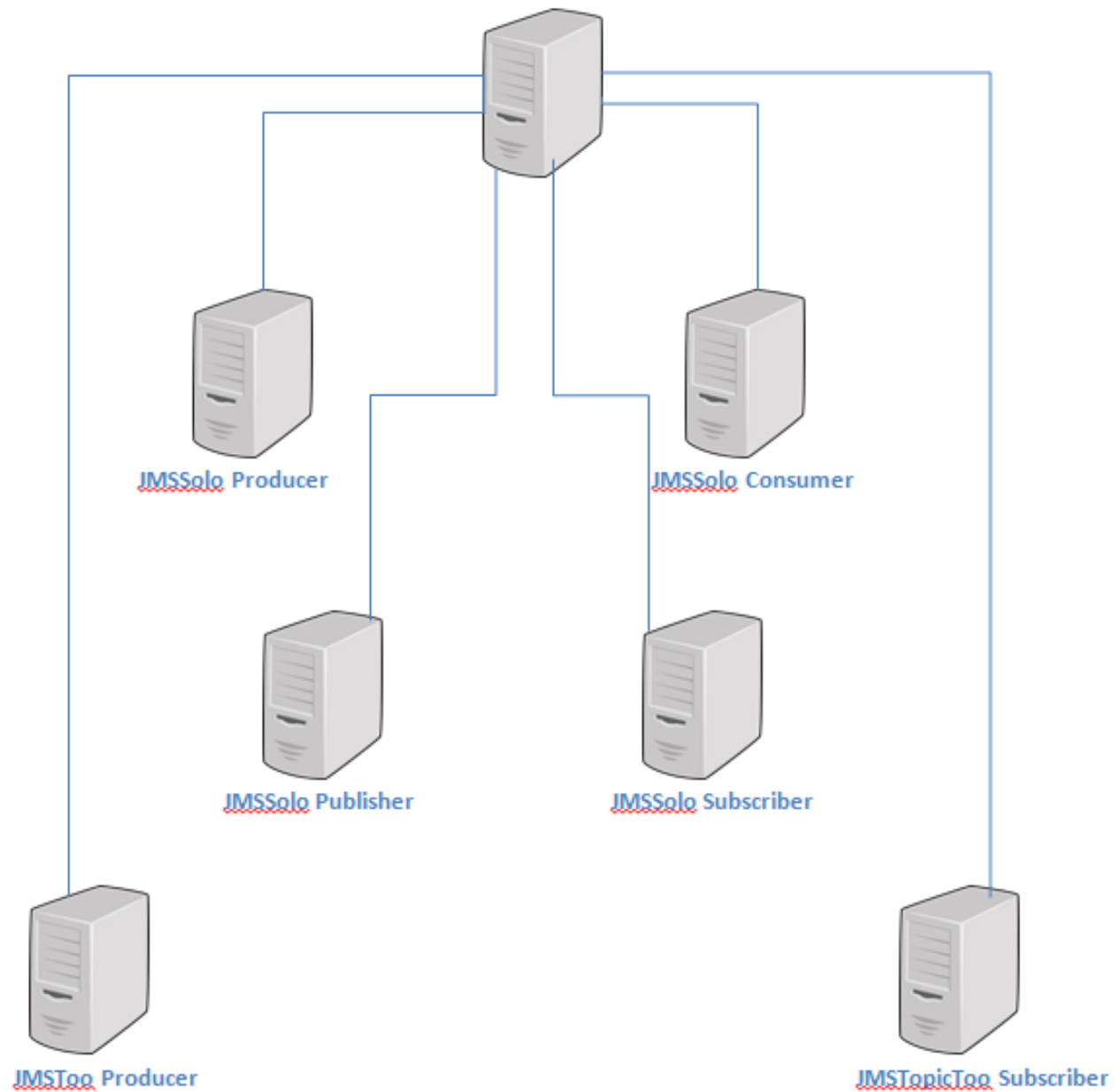
```
<jms:listener destination="mumEATopic" ref="PubSubMessageListener"
```


JMS DEMO



Distributed Server View of JMS Demo

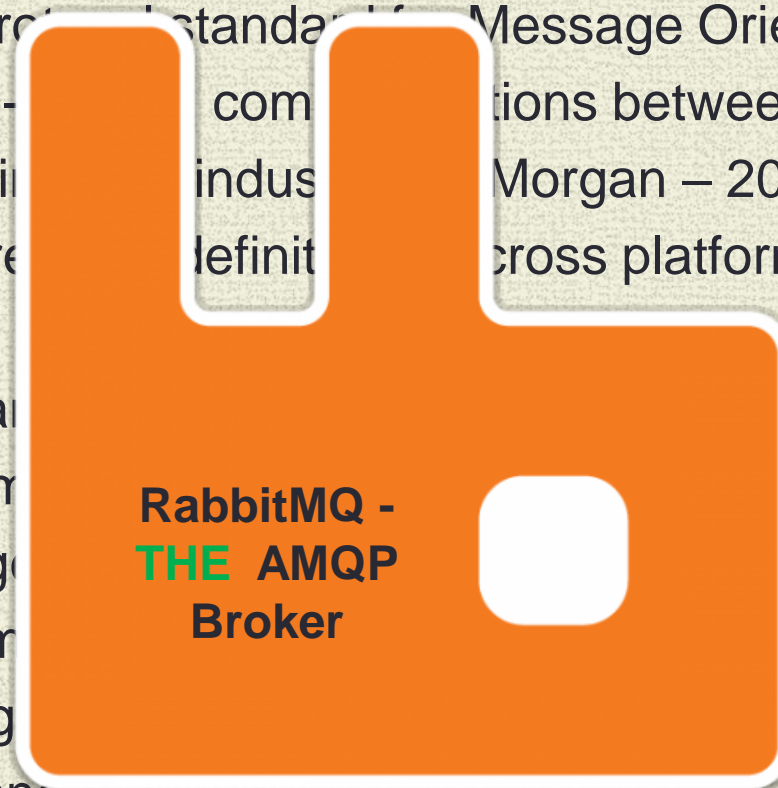
ActiveMQ MOM Server



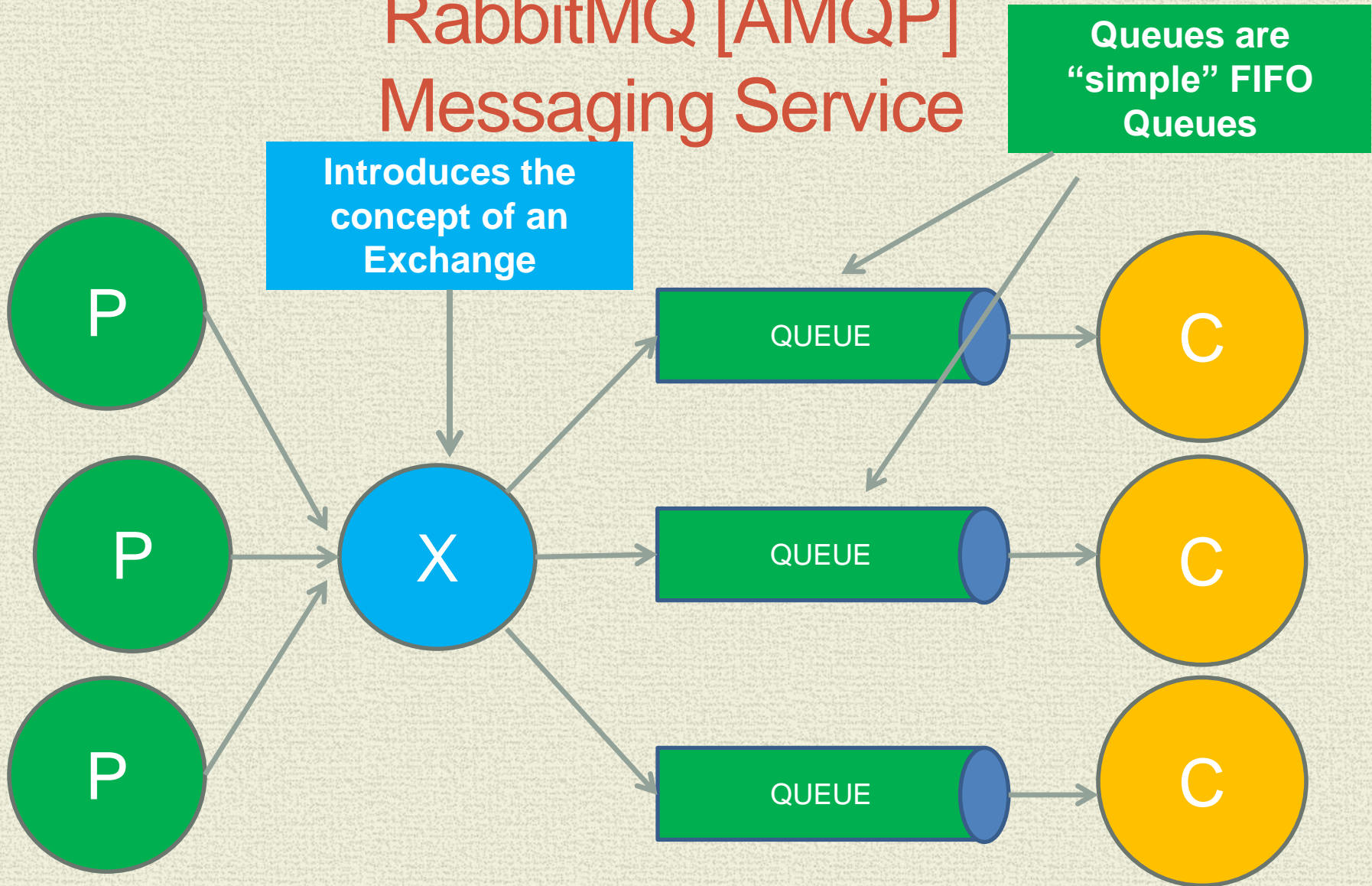
AMQP

(Advanced Message Queuing Protocol)

- AMQP is an open protocol standard for Message Oriented Middleware
For queue-based communications between applications
- Developed for the financial industry [Morgan – 2006]
- AMQP defines a wire format for cross platform interoperability
- All AMQP clients can communicate with any AMQP servers
- Diverse programs can connect to AMQP and integrate easily
- Legacy message protocols can be migrated to AMQP to remove proprietary protocols from the system
- Enables message-based architectures
- PTP & publish-and-subscribe messaging
- Transactional messaging functionality



RabbitMQ [AMQP] Messaging Service



Producer .vs. Consumer Centric Queues

Queues are only consumed by a single receiver

Producer Centric queue

If more than one consumer subscribes to the queue, the messages are dispensed in a round-robin fashion.

AKA a work queue

Consumer defined queue[s]

Multiple queues, bound to the same exchange/routing key

Emulate a broadcast message

“A queue per consumer”

See AmqpClient Demo

AMQP Concepts

- **Exchanges** - Message routing agents; accept messages from producers routes to queues
- **Bindings** - binds/maps a queue & exchange
- **Routing Key** - optional attribute to customize binding/routing
- **Queues** - Message placeholders

AMQP Exchanges

- **Direct** - Queue binding requires a direct match based on a “simple” Routing Key. Corresponds to JMS PTP.
NOTE: can have multiple Queues/Consumers
- **Fanout** - Queue is bound directly to exchange no Routing Key. Corresponds to “basic” JMS Pub/Sub.
- **Topic** - Queue binding requires a direct match based on a “complex” Routing Key [*beyond basic JMS Pub/Sub*]
- **Headers** - Similar to Topic only uses message headers instead of explicit Routing Key
- Direct & Fanout are identified as **MANDATORY** by AMQP

Topic Demo Use Case

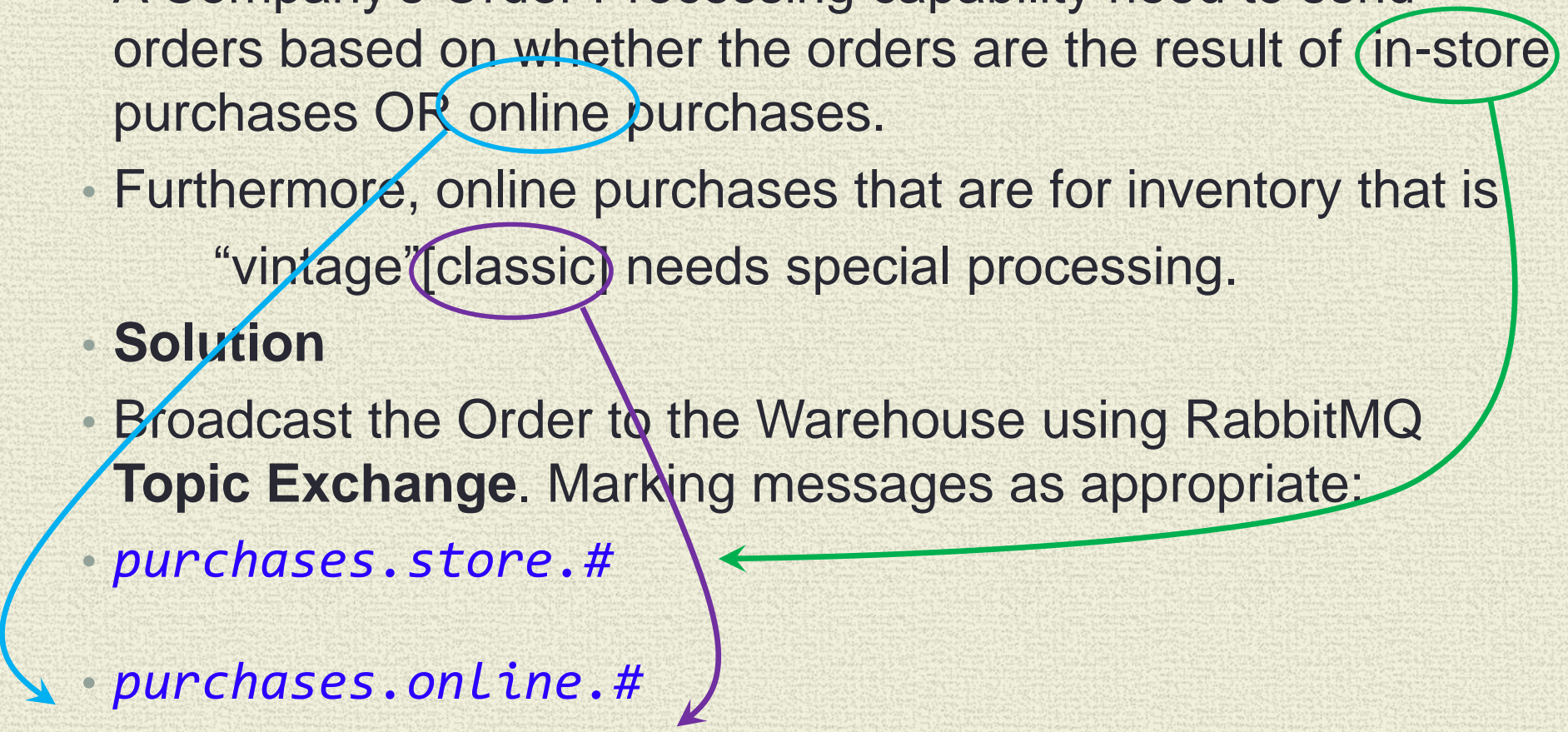
- **Use Case**

- A Company's Order Processing capability need to send orders based on whether the orders are the result of **in-store** purchases OR **online** purchases.
- Furthermore, online purchases that are for inventory that is "vintage"**[classic]** needs special processing.

- **Solution**

- Broadcast the Order to the Warehouse using RabbitMQ **Topic Exchange**. Marking messages as appropriate:

- *purchases.store.#*
- *purchases.online.#*
- *purchases.online.classic.#*



Topic Exchange Routing key

Consists of a list of attributes

Delimited by periods [“.”]

Up to 255 characters in length

- Wild cards:
- * (asterisk) can substitute for exactly one word.
- # (hash) can substitute for zero or more words.

• Examples:

```
<rabbit:binding queue="purchasesStore" pattern="purchases.store.#" />
<rabbit:binding queue="purchasesOnline" pattern="purchases.online.#" />
<rabbit:binding queue=" purchasesOnlineClassic"
                pattern="purchases.online.classic.#" />
```

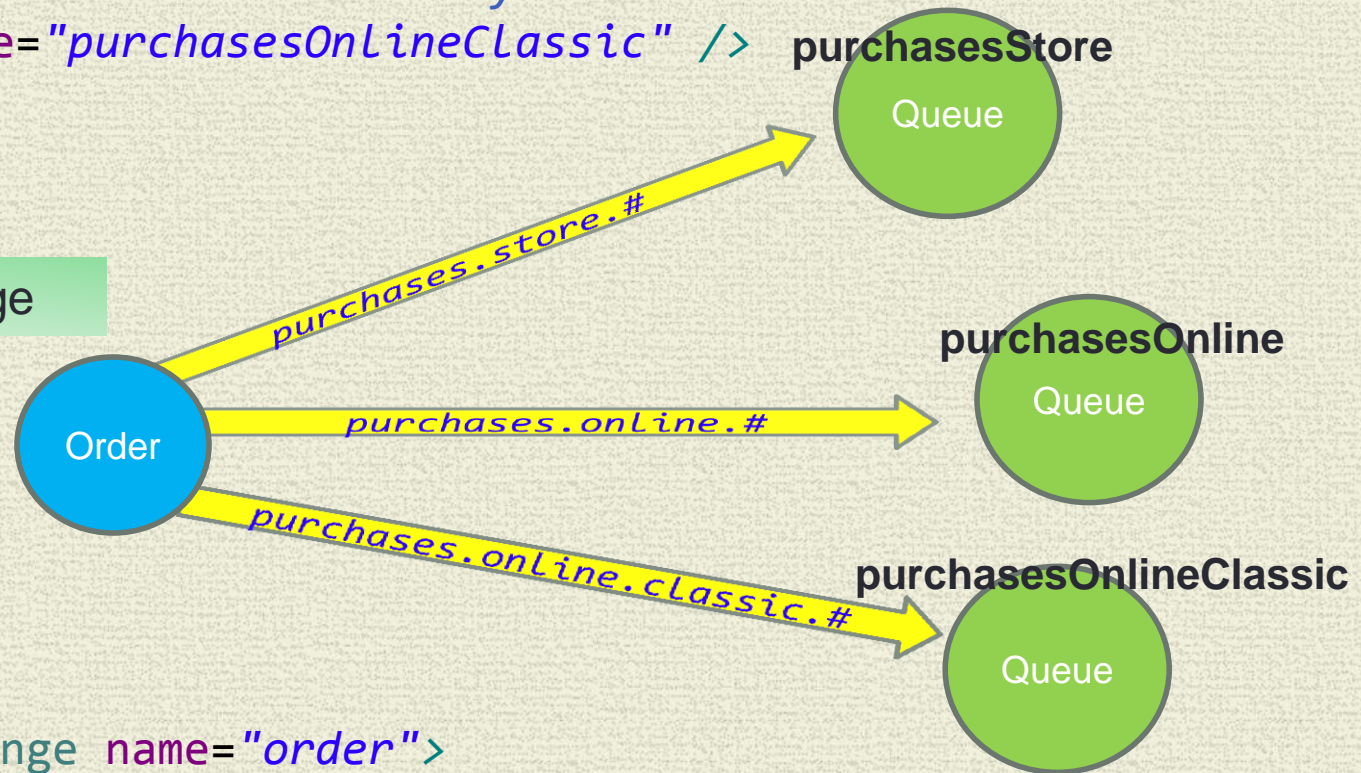
Routing Key



DEMO - Topic Exchange – Order

```
<rabbit:queue name="purchasesStore" />
<rabbit:queue name="purchasesOnline" />
<!-- added topic filter to bind only orders that are "classic" -->
<rabbit:queue name="purchasesOnlineClassic" />
```

Bind Queue to Exchange



```
<rabbit:topic-exchange name="order">
<rabbit:binding queue="purchasesOnline" pattern="purchases.online.#"
<rabbit:binding queue="purchasesStore" pattern="purchases.store.#"
<rabbit:binding queue="purchasesOnlineClassic"
pattern="purchases.online.classic.#"
```


Topic Producer: Demo Topic Exchange

```
<rabbit:template id="topicTemplate" connection-factory= "connectionFactory"
                    routing-key="purchases.store" exchange="order" />
```

Producer code : OrderServiceImpl.java

```
• public class OrderServiceImpl implements OrderService {
•     public void publish(RabbitTemplate rabbitTemplate) {
        rabbitTemplate.convertAndSend("purchases.online.classic",order)
        rabbitTemplate.convertAndSend("purchases.store",order);
        rabbitTemplate.convertAndSend("purchases.online",order);
    }
}
```

Topic Consumer Example:

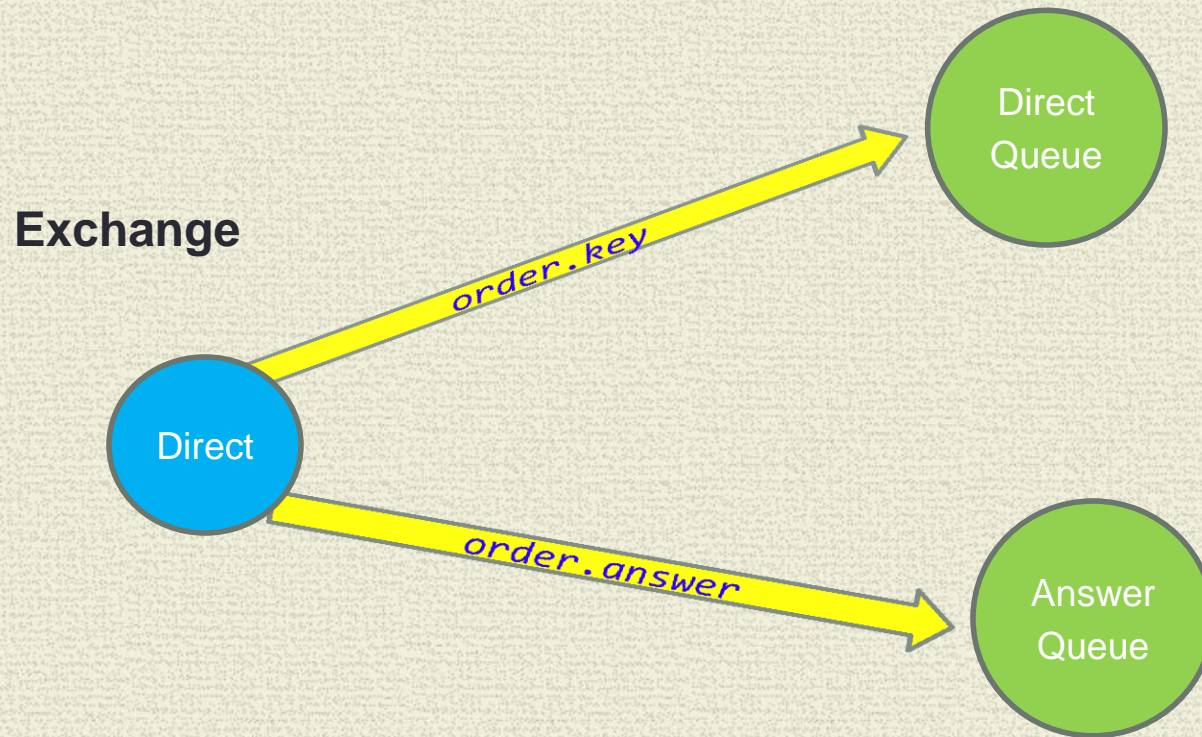
```
<rabbit:listener ref="orderListener" method="listen"
                queue-names="purchasesOnlineClassic" />
<bean id="orderListener" class="edu.mum.amqp.OrderListener" />
```

Consumer code : OrderListener.java

```
public class OrderListener {
    public void listen(Order order) {
    }
}
```


DEMO - Direct Exchange

```
<rabbit:queue name="directQueue" />  
<rabbit:queue name="answerQueue" />
```



```
<rabbit:direct-exchange name="direct">  
<rabbit:binding queue="directQueue" key="order.key">  
<rabbit:binding queue="answerQueue" key="order.answer">
```


Demo Direct Exchange

Direct Producer:

```
<rabbit:template id="directTemplate" connection-factory= "connectionFactory"
                routing-key="order.key" exchange="direct" />
```

Producer code : DirectServiceImpl.java

- `public class DirectServiceImpl implements OrderService {`
- `public void publish(RabbitTemplate rabbitTemplate) {`
 `rabbitTemplate.convertAndSend(orderItem)`

Direct Consumer Example:

```
<rabbit:listener ref="queueListener" method="listen"
                queue-names="directQueue" />
<bean id="queueListener" class="edu.mum.amqp.DirectListener" />
```

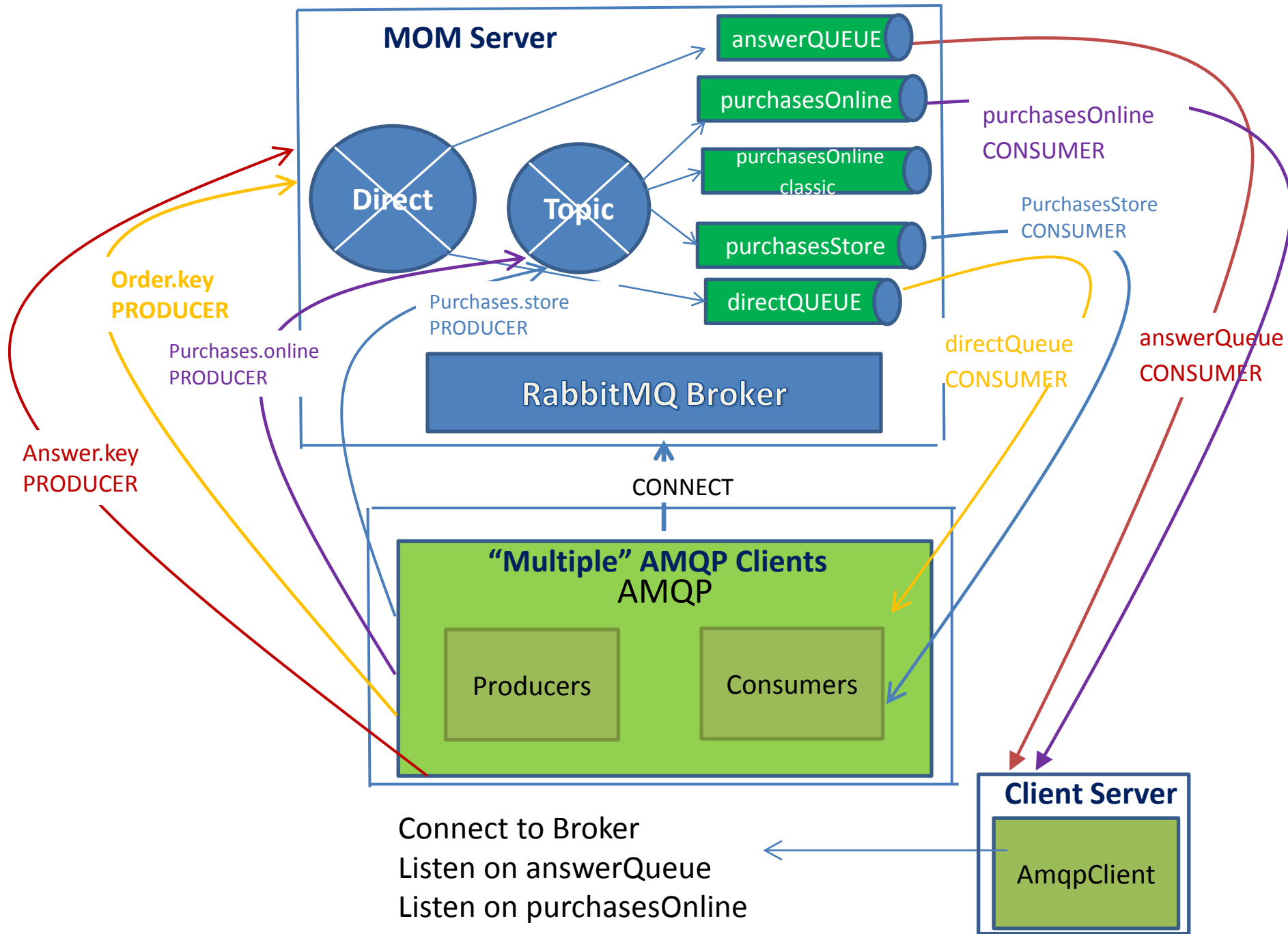
- Consumer code : DirectListener.java

```
public class DirectListener {
    public void listen(Order order) {
```


DEMO – invoking the Producers

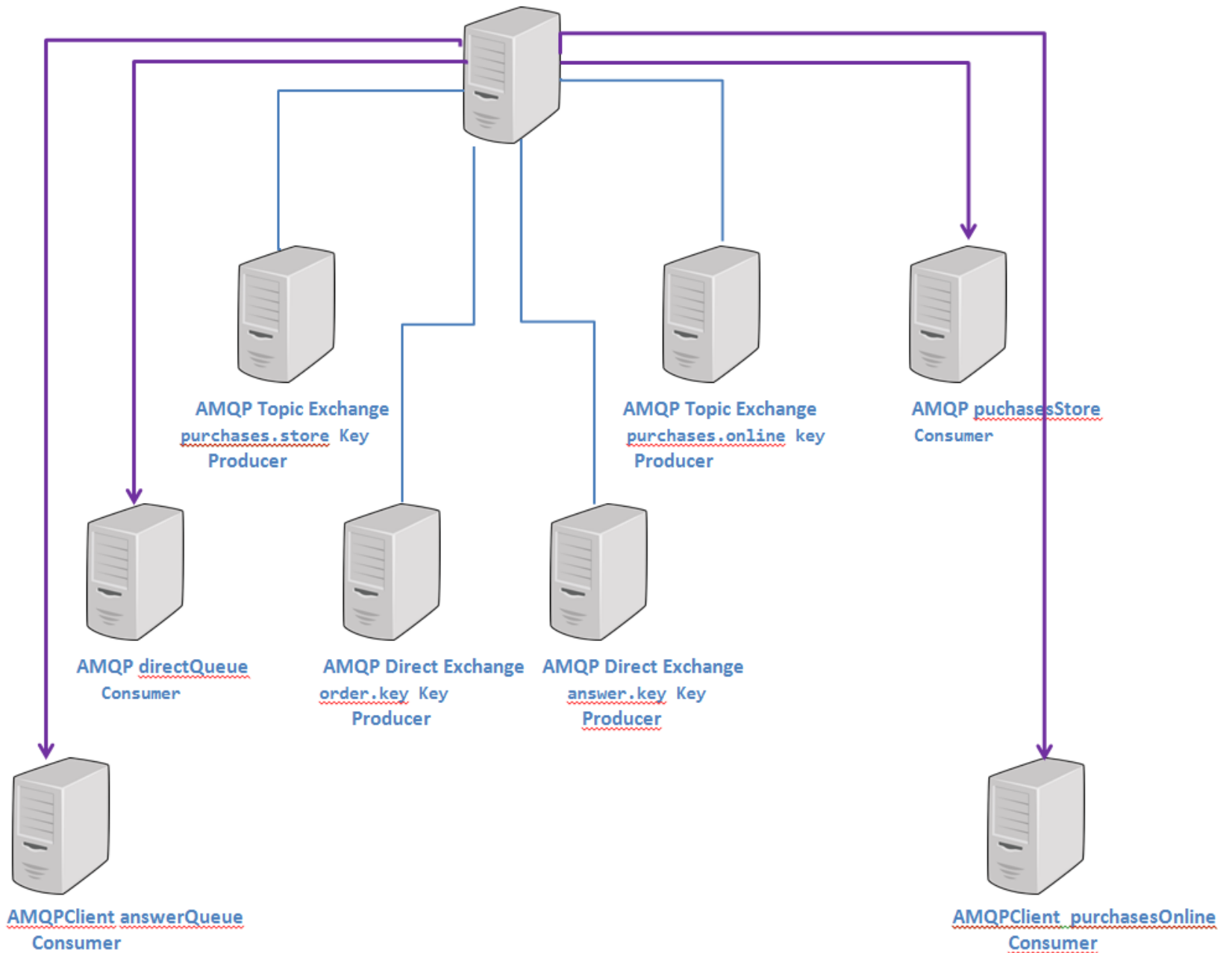
- `public static void main(String[] args) {`
-
- `// Publish to Topic`
- `RabbitTemplate topicTemplate =`
- `context.getBean("topicTemplate", RabbitTemplate.class);`
- `OrderService orderService = new OrderServiceImpl();`
- `orderService.publish(topicTemplate);`
- `// Publish to "direct" exchange on order.key == directQueue`
- `RabbitTemplate directTemplate =`
- `context.getBean("directTemplate", RabbitTemplate.class);`
- `OrderService directService = new DirectServiceImpl();`
- `directService.publish(directTemplate);`

AMQP DEMO



Distributed Server View of AMQP Demo

RabbitMQ MOM Server



Main Point

- AMQP integrates heterogeneous systems through the introduction of a basic wire protocol.
- ***Science of Consciousness:*** *Acting from the level of Transcendental Consciousness, thoughts and actions are more integrated and harmonious*

