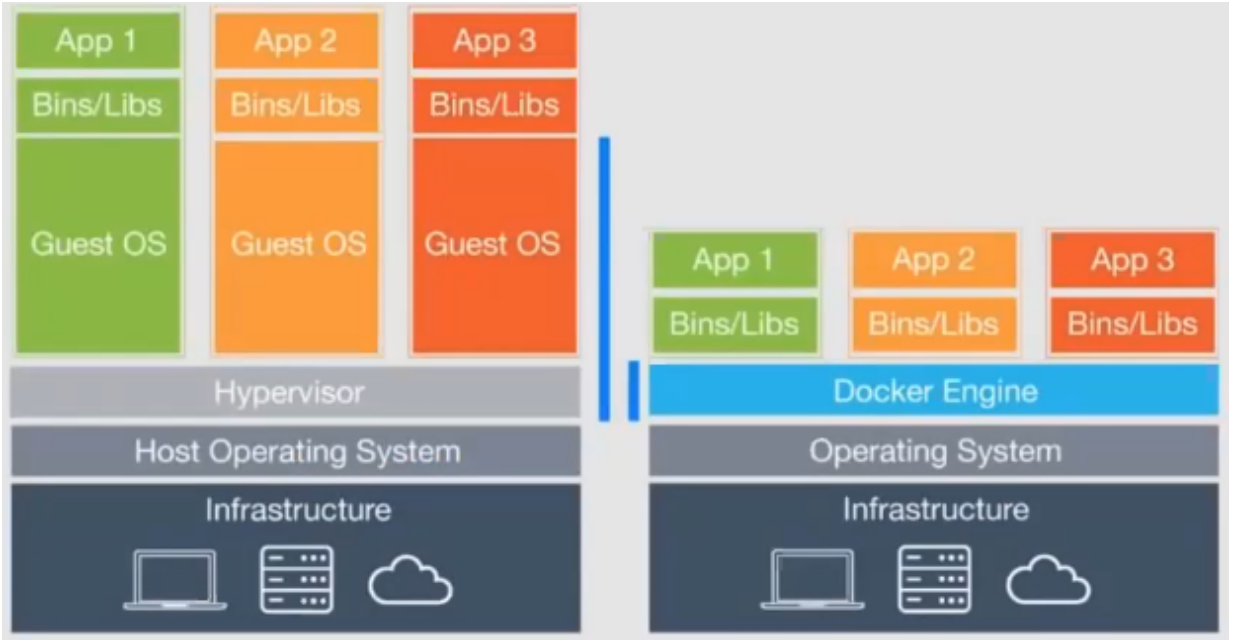
LAB 3 - JMS (& Docker)

Docker
Each virtual machine includes the application, the necesary binaries to run and an entire guest operating system.
Containers have similar resource isolation and allocation benefits as virtual machines, but have a different architectural approach that allows them to be much more portable and efficient;
Containers (docker) running on a single machine share the same operating system kernel, so they can start instantly and make more efficeint use of RAM. Images are constructed from layered filesystems so the can share common files, making disk usage and image downloads much more efficient.



Docker allows you to package an application with all of its dependencies into a standardized unit for software development, wrapped up in a complete file system that contains everything needed to run the application, so it guaranteed that the application will always run in the same ambient, even when the hardware and the OS changes. Containers use the docker abstracted system resources and have different layers, to share common files with other containers. Typically, when designing an application or service using Docker, it's best to break out the functionalities into individual containers (micro-service architecture). This allows to easily scale, update or replace components independently in the future.
Advantages.

- Containers are lighter than VMs
- Portability, all of the dependencies for an application are bundled inside the container, allowing it to run on any host
- Predictability: the container is isolated and the application is accessible only through the standardized expose interfaces, so it will run the same way regardless of the host.

The docker engine is a client-server program composed by a daemon with REST APIs and a command line interface which is used to give commands to the daemon through the exposed

APIs. The docker daemon can run either on the same system or on a remote system and it builds, distributes and runs the containers.

To build custom images we can create a container and use it to build our application manually or automatize the installation of the application and its requirements with a dockerfile.
A Dockerfile defines what goes on in the environment inside your container. Access to resources like networking interfaces and disk drives is virtualized inside this environment, which is isolated from the rest of your system, so you need to map ports to the outside world, and be specific about what files you want to "copy in" to that environment. However, after doing that, you can expect that the build of your app defined in this Dockerfile behaves exactly the same wherever it runs.
Some of the common commands that we can use in a Dockerfile are:

- FROM: sets the parent image of the current image
- WORKDIR: sets the working directory
- COPY: copies file and directories into the container
- RUN: runs console commands
- EXPOSE: opens ports from container, to reach them from the outside
- ENV: defines environment variables
- CMD: commands to be run when the container starts

Docker Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a Compose file to configure your application's services. Then, using a single command, you create and start all the services from your configuration.
Using compose is a 3 step process

1. Define the app environment with a dockerfile so it can be reproduced
2. Define the services that make up your app in a docker-compose.yml so they can be run together in an isolated environment
3. Run docker-compose up and it will start and run the entire app

JMS

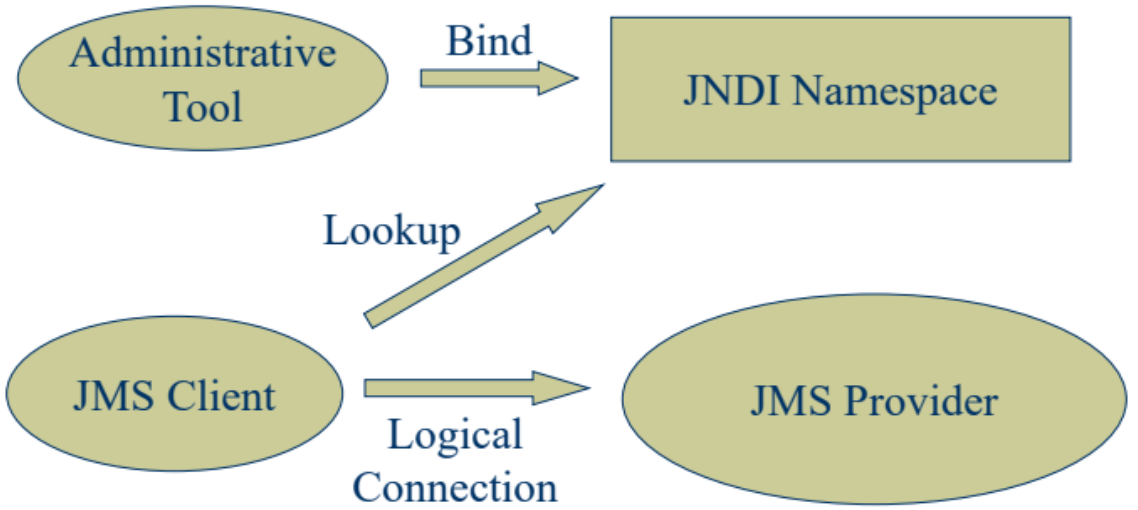
Java Messaging Service, Message oriented middleware. More precisely, a specification/API that describes a common way for java programs to create, send, receive and read distributed messages. It's based on loosely coupled communication, asynchronous messaging and reliable delivery. A JMS application is composed of (architecture):

- JMS Clients: Java programs that send/receive messages
- Administered Objects: preconfigured JMS objects created by an admin for the use of the clients such as ConnectionFactory (used by a client to establish a connection to the JMS provider), Destination (queue or topic). These allow for portability because they hide the JMS Provider details.
- JMS Provider: messaging system that implements JMS and its administrative functionalities.
- Messages

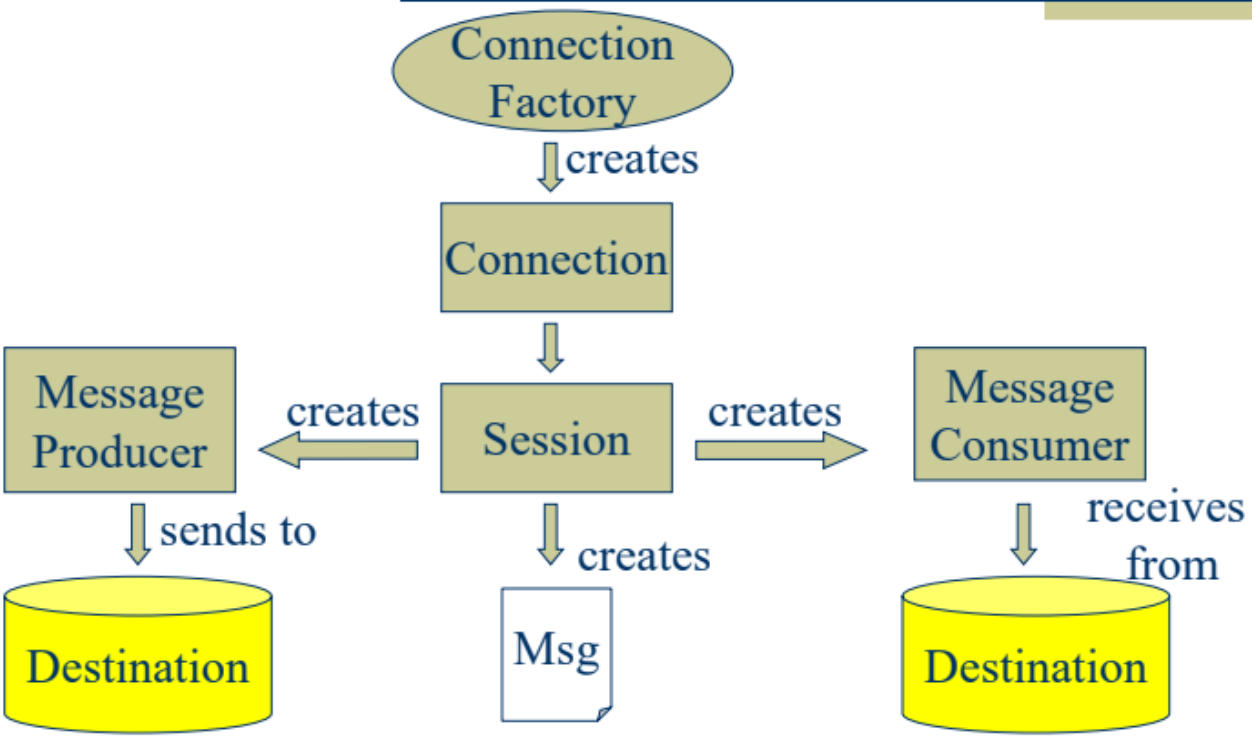
It supports both Point-To-Point (message queues with only one consumer) and Publish-Subcribe systems (uses a "topic" to send and receive messages. which has multiple consumers)
Message consumption can happen:

- 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

The administered objects (ConnectionFactory, Destination) are placed in a JNDI (Java Naming & Directory Interface) namespace under well-known identifiers. To establish a connection to a JMS provider, clients retrieves a *ConnectionFactory* from the JNDI namespace.



JMS API Programming Model



A *Connection* represents a link between the client and a JMS Provider
A *Destination* encapsulates the identity of a message destination
A *Session* is a single-threaded context for sending and receiving messages
Message producers and *consumers* send and receive messages in a *Session* respectively.

JMS Messages are in the form:

- Message Header
  - Used for identifying and routing messages
  - Contains vendor-specified values, but could also contain app-specific data
  - typically name/value pairs
- Message Properties (Optional)

- Message Body
  - Contains the data
  - One of the 5 different message body types:

Message Type	Contains	Some Methods
TextMessage	String	getText,setText
MapMessage	set of name/value pairs	setString,setDouble,setLong,getDouble,getString
BytesMessage	stream of uninterpreted bytes	writeBytes,readBytes
StreamMessage	stream of primitive values	writeString,writeDouble,writeLong,readString
ObjectMessage	serialize object	setObject,getObject

A JMS Client will typically:

- Use JNDI to find a *ConnectionFactory* object and one or more *Destination* obects.
- Use the *ConnectionFactory* to establish a *Connection* to the JMS PRovider
- User the *Connection* to create one or more *Sessions*
- Combine *Sessions* and *Destinations* to create the needed *MessageConsumer* and *MessageProducer*
- Tell the *Connection* to start the delivery of messages

So there are a total of 6 "actors" : ConnectionFactory, Destination, Sessions, MessageConmsumer, MessageProducer, Connection.

Concurrency

JMS restricts concurrent access to Destination, ConnectionFactory and Connection;
Session, MessageProducer and MessageConsumer would require very complex multithreading and concurrency support. More over, a Session can either use synchronous receive or asynchronous (not both).

Message Ordering and Acknowledgment

- FIFO between a given sender and receiver
- If a Session is "transacted", message acks are handled automatically, if it's not transacted, one can set DUPS_OK_ACKNOWLEDGE, AUTO_ACKNOWLEDGE or CLIENT_ACKNOWLEDGE to handle acks in different ways with different overhead.

Message Delivery Mode

- NON_PERSISTENT: does not require logging to stable storage and uses minimum overhewad. A JMS Provider's failure can cause a message to be lost. The message is delivered at most once.

- PERSISTENT: instructs the JMS Provider to take extra care to guarantee message delivery. A message is delivered only once.

PTP Domain Example

```
import javax.jms.*; import javax.naming.*;

public class Sender {
  static Context ictx = null;
  public static void main(String[] args) throws
     Exception {
    ictx = new InitialContext();
    Queue queue = (Queue) ictx.lookup("queue");
    QueueConnectionFactory qcf =
     (QueueConnectionFactory) ictx.lookup("qcf");
    ictx.close();

    QueueConnection qc = qcf.createQueueConnection();
    QueueSession qs = qc.createQueueSession(false,
                      Session.AUTO_ACKNOWLEDGE);
    QueueSender qsend = qs.createSender(queue);
    TextMessage msg = qs.createTextMessage();

    int i;
    for (i = 0; i < 10; i++) {
      msg.setText("Test number " + i);
      qsend.send(msg);
    }

    qc.close();
  }
}
```

```
import javax.jms.*; import javax.naming.*;

public class Receiver {
  static Context ictx = null;
  public static void main(String[] args) throws
     Exception {
    ictx = new InitialContext();
    Queue queue = (Queue) ictx.lookup("queue");
    QueueConnectionFactory qcf =
     (QueueConnectionFactory) ictx.lookup("qcf");
    ictx.close();

    QueueConnection qc = qcf.createQueueConnection();
    QueueSession qs = qc.createQueueSession(false,
                      Session.AUTO_ACKNOWLEDGE);
    QueueReceiver qrec = qs.createReceiver(queue);
    TextMessage msg;

    qc.start();

    int i;
    for (i = 0; i < 10; i++) {
      msg = (TextMessage) qrec.receive();
      System.out.println("Msg received: " +
        msg.getText());
    }

    qc.close();
  }
}
```

Publish-Subscribe Domain Example

```
import javax.jms.*; import javax.naming.*;

public class Publisher {
  static Context ictx = null;
  public static void main(String[] args) throws Exception {
    ictx = new InitialContext();
    Topic topic = (Topic) ictx.lookup("topic");
    TopicConnectionFactory tcf = (TopicConnectionFactory)
     ictx.lookup("tcf");
    ictx.close();

    TopicConnection tc = tcf.createTopicConnection();
    TopicSession ts = tc.createTopicSession(true,
                      Session.AUTO_ACKNOWLEDGE);
    TopicPublisher tpub = ts.createPublisher(topic);
    TextMessage msg = ts.createTextMessage();

    int i;
    for (i = 0; i < 10; i++) {
      msg.setText("Test number " + i);
      tpub.publish(msg);
    }

    ts.commit();
    tc.close();
  }
}
```

```
import javax.jms.*; import javax.naming.*;

public class Subscriber {
  static Context ictx = null;
  public static void main(String[] args) throws Exception {
    ictx = new InitialContext();
    Topic topic = (Topic) ictx.lookup("topic");
    TopicConnectionFactory tcf = (TopicConnectionFactory)
     ictx.lookup("tcf");
    ictx.close();

    TopicConnection tc = tcf.createTopicConnection();
    TopicSession ts = tc.createTopicSession(true,
                      Session.AUTO_ACKNOWLEDGE);
    TopicSubscriber tsub = ts.createSubscriber(topic);

    tsub.setMessageListener(new MsgListener());
    tc.start();
    System.in.read();
    tc.close();
  }
}
class MsgListener implements MessageListener {
  String id;
  public MsgListener() {id = "";}
  public MsgListener(String id) {this.id = id;
  }
  public void onMessage(Message msg) {
    TextMessage tmsg = (TextMessage) msg;
    try {
      System.out.println(id+": "+tmsg.getText());
    } catch (JMSException jE) {
      jE.printStackTrace();
    }
  }
}
```

RabbitMQ is our JMS Provider.
The java file in our example is a publisher.
The Servant has to do 2 main tasks: Produce quotations for the stocks and notify of a successful or unsuccessful purchase, so it has a class for each of service, plus a main class.

StockMarketServant

The main class. It only has a main, whose duty is to launch both services.

```
public class StockMarketServant {

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

	NotificatoreAcquisto n = new NotificatoreAcquisto();
		n.start();

	    ProduttoreQuotazioni q = new ProduttoreQuotazioni();
		q.start();


	}
}
```

ProduttoreQuotazioni

It' s a publisher, so has the general skeleton of a jms publisher in the start method.

- **scegliTitolo()**: It's a method that chooses a random name among an array of names (the array is instantiated outside the method).

```
final String titoli[] = { "Telecom", "Finmeccanica", "Banca_Intesa",
		"Oracle", "Parmalat", "Mondadori", "Vodafone", "Barilla" };

private String scegliTitolo() {
	int whichMsg;
	Random randomGen = new Random();

	whichMsg = randomGen.nextInt(this.titoli.length);
	return this.titoli[whichMsg];
}
```

- **valore():** generates a random float value to associate to a stock name.

```
private float valore() {
	Random randomGen = new Random();
	float val = randomGen.nextFloat() * this.titoli.length * 10;
	return val;
}
```

- **start() throws NamingException, JMSException:**method called by main to start publishing and uses the common interfaces. It starts by declaring the needed variables (context, ConnectionFactory, Connection, Session, Destination, MessageProducer and a destination name), setting the properties and creating the JNDI InitialContext Object.

```
private static final Logger LOG = LoggerFactory.getLogger(ProduttoreQuotazioni.class);

public void start() throws NamingException, JMSException {

        Context jndiContext = null;
        ConnectionFactory connectionFactory = null;
        Connection connection = null;
        Session session = null;
        Destination destination = null;
        MessageProducer producer = null;
        String destinationName = "dynamicTopics/Quotazioni";

        /*
 * Create a JNDI API InitialContext object
 */

try {
    Properties props = new Properties();
    props.setProperty(Context.INITIAL_CONTEXT_FACTORY,"org.apache.activemq.jndi.ActiveMQInitialContextFactory
    props.setProperty(Context.PROVIDER_URL,"tcp://localhost:61616");
    jndiContext = new InitialContext(props);
} catch (NamingException e) {
    LOG.info("ERROR in JNDI: " + e.toString());
    System.exit(1);
}

/*
 * Look up connection factory and destination.
 */
try {
    connectionFactory = (ConnectionFactory)jndiContext.lookup("ConnectionFactory");
    destination = (Destination)jndiContext.lookup(destinationName);
} catch (NamingException e) {
    LOG.info("JNDI API lookup failed: " + e);
    System.exit(1);
}
```

- Then it looks up the ConnectionFactory and destination

```
try {
    connectionFactory = (ConnectionFactory)jndiContext.lookup("ConnectionFactory");
    destination = (Destination)jndiContext.lookup(destinationName);
} catch (NamingException e) {
    LOG.info("JNDI API lookup failed: " + e);
    System.exit(1);
}
```

- Then create the Connection, from it create the Session (false means the session is not transacted). Then create the sender and text message.

```
try {
    connection = connectionFactory.createConnection();
    session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    producer = session.createProducer(destination);



        TextMessage message = null;
        String messageType = null;

        message = session.createTextMessage();
```

- The messages are created and sent in a while loop using the 2 methods above. The properties and text are set and the message is sent.

```
float quotazione;
int i = 0;
while (true) {
        i++;
        messageType = scegliTitolo();
        quotazione = valore();
        message.setStringProperty("Nome", messageType);
        message.setFloatProperty("Valore", quotazione);
        message.setText(
                        "Item " + i + ": " + messageType + ", Valore: "
                        + quotazione);

            LOG.info(
                        this.getClass().getName() +
                        "Invio quotazione: " + message.getText());

        producer.send(message);

        try {
                Thread.sleep(5000);
        } catch (Exception ex) {
                ex.printStackTrace();
        }
}
```

- Finally close the connection

```
catch (JMSException e) {
    LOG.info("Exception occurred: " + e);
} finally {
    if (connection != null) {
        try {
            connection.close();
        } catch (JMSException e) {
        }
    }
}
```

NotificatoreAcquisto

It's both a subscriber and a publisher. this time uses a publish-subscribe domain.

- It starts with setting up the required variables

```
public class NotificatoreAcquisto implements MessageListener {

        private static final org.slf4j.Logger LOG = LoggerFactory.getLogger(NotificatoreAcquisto.class);

        Properties properties = null;
        Context jndiContext = null;
        private TopicConnectionFactory connectionFactory = null;
        private TopicConnection connection = null;
        private TopicSession session = null;
        private Topic destination = null;
        private TopicSubscriber subscriber = null;
        private TopicPublisher publisher = null;

        private Random randomGen = new Random();
```

- **start():** completes the JMS routine and sets up both a subscriber and a publisher, starting the connection and calling subscirber.setMessageListener(this).

```
public void start() throws NamingException, JMSException {

    InitialContext ctx = null;

    try {
        properties = new Properties();
        properties.setProperty(Context.INITIAL_CONTEXT_FACTORY,"org.apache.activemq.jndi.Active
        properties.setProperty(Context.PROVIDER_URL,"tcp://localhost:61616");
        jndiContext = new InitialContext(properties);
    } catch (NamingException e) {
        LOG.info("ERROR in JNDI: " + e.toString());
        System.exit(1);
    }

    ctx = new InitialContext(properties);
    this.connectionFactory = (TopicConnectionFactory) ctx.lookup("ConnectionFactory");
    this.destination = (Topic) ctx.lookup("dynamicTopics/Ordini");
    this.connection = this.connectionFactory.createTopicConnection();
    this.session = this.connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
    this.subscriber = this.session.createSubscriber(this.destination, null, true);
    this.publisher = this.session.createPublisher(this.destination);
    this.connection.start();

    Logger.getLogger(this.getClass().getName()).info("In attesa di richieste di acquisto...");

    subscriber.setMessageListener(this);
}
```

- **onMessage()**: callback method for the subscriber. First it catches the message and gets its properties. Then it creates a topicSession, a publisher and a message using the properties it got. Then sends the new message.

```
public void onMessage(Message mex) {
        TextMessage message;
        String utente = null;
        String nome = null;
        float prezzo;
        int quantita;
        boolean status = randomGen.nextFloat() < 0.5;
        try {
                message = (TextMessage) mex;
                utente = message.getStringProperty("Utente");
                nome = message.getStringProperty("Nome");
                prezzo = message.getFloatProperty("Prezzo");
                quantita = message.getIntProperty("Quantita");
        } catch (Exception e) {
                e.printStackTrace();
                return;
        }
        try {
                session = connection.createTopicSession(false,
                                Session.AUTO_ACKNOWLEDGE);
                publisher = session.createPublisher(destination);
                message = session.createTextMessage();
                message.setStringProperty("Utente", utente);
                message.setStringProperty("Nome", nome);
                message.setBooleanProperty("Status", status);
                message.setIntProperty("Quantita", quantita);
                message.setFloatProperty("Prezzo", prezzo);

                Logger.getLogger(
                                this.getClass().getName()
                        ).info(
                                "********************************************" + "\n" +
                                "Notifica richiesta di acquisto" + "\n" +
                                "ID utente: " + utente + "\n" +
                                "Titolo: " + nome + "\n" +
                                "Quantit\u00e0: " + quantita + "\n" +
                                "Prezzo: " + prezzo + "\n" +
                                "Accettato: " + status + "\n" +
                                "********************************************"
                        );

                publisher.send(message);
        } catch (Exception err) {
                err.printStackTrace();
        }
}
```

dont ignore me bitch