LAB1 – Soap Web Services.

The idea of this exercise is to be able create a web services that's able to offer functionalities to the user, and at same time create a client that's able to use the one created by us and one offered by the professor.

The Web Service provided by the prof will be a **BankInterface** (wsdl based) offering the following operation: **java.lang.String[] getOperationsByClientID (int ClientID)** that, given a Client ID will return a list with all the ID of all the operation performed by that specific client as an array of string in the form of:
"[ID,ID of the performing client,date,amount, description]"

result = **getOperationsByClientID (1)**
result[0] = 4,1,2015-06-01,150,Cena al ristorante
result[1] = 5,1,2015-06-02,200,Regalo amante
…
..

We are requested to:
1)Create and run an AAAWS web service that offer the following operation.
java.lang.String[] getClients(): that returns all the IDs and names of the clients stored in the database as an array of string, each of them in the form of:
"[ID of the client, FirstName Lastname]", eg => ["1,Massimo Mecella" "2,Miguel Ceriani"].

2)Create a client program that output all of the client that have performed an operation with description ("benzina autostrada").

//

The AAAWS Web Service

New Project -> Java With Maven -> Java Application

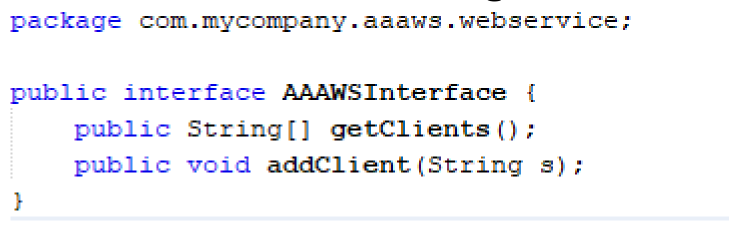
What we need to create? We for sure need a ServerInterface and a ServerImplementation.

1.1)AAAWSInterface:

RightClick Package -> New Java Interface

IMPORTANT: before the "publicInterface classname(){}" you need to add a

@WebService

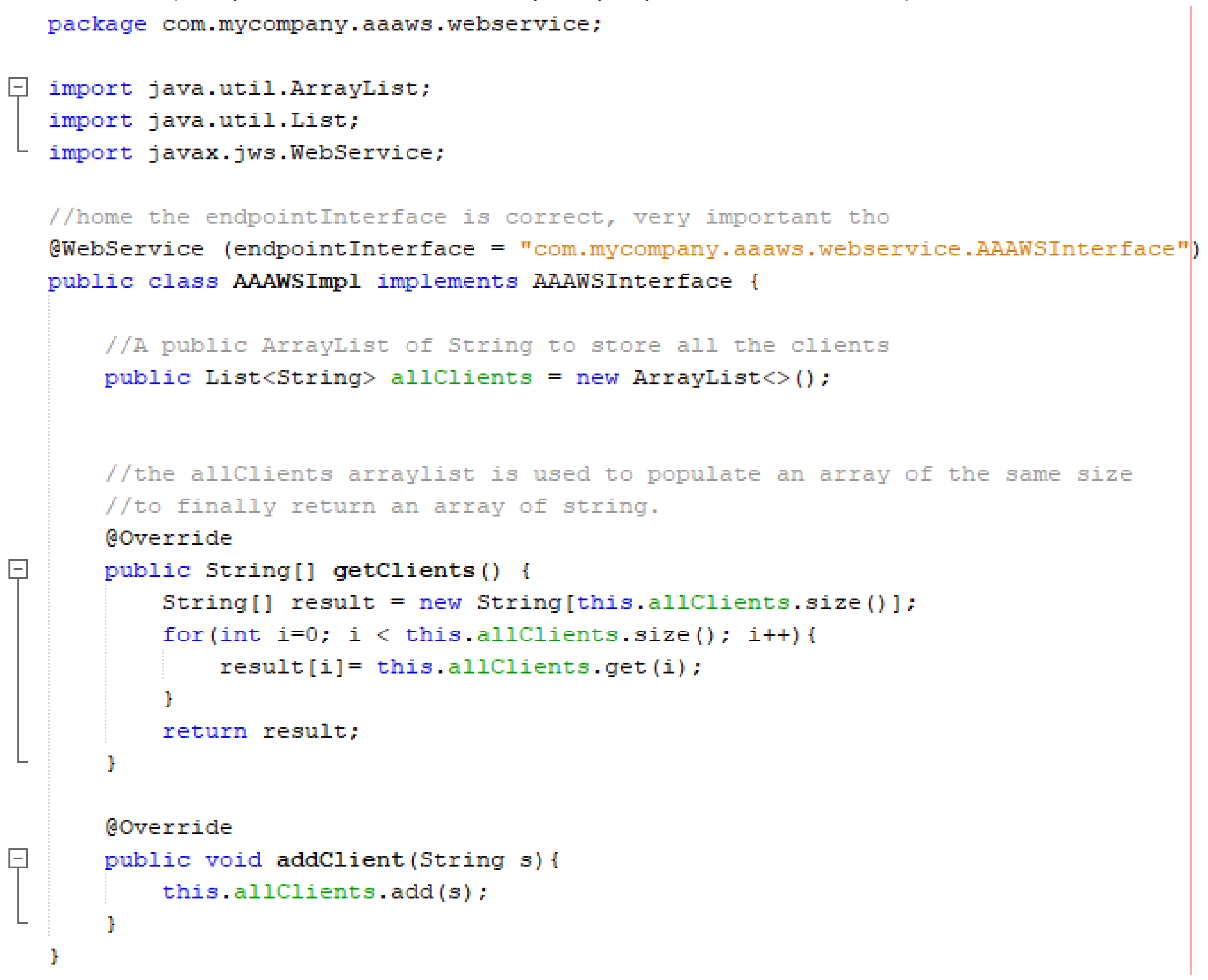


```
package com.mycompany.aaaws.webservice;

public interface AAAWSInterface {
    public String[] getClients();
    public void addClient(String s);
}
```

1.2)AAAWSImpl

RightClick Package -> New Java Class

Important, this time before the implemented methods we have to add something more:
@WebService(endpointInterface = “com.mycompany.aaaws.webservice”)



```java
package com.mycompany.aaaws.webservice;

import java.util.ArrayList;
import java.util.List;
import javax.jws.WebService;

//home the endpointInterface is correct, very important tho
@WebService (endpointInterface = "com.mycompany.aaaws.webservice.AAAWSInterface")
public class AAAWSImpl implements AAAWSInterface {

    //A public ArrayList of String to store all the clients
    public List<String> allClients = new ArrayList<>();


    //the allClients arraylist is used to populate an array of the same size
    //to finally return an array of string.
    @Override
    public String[] getClients() {
        String[] result = new String[this.allClients.size()];
        for(int i=0; i < this.allClients.size(); i++){
            result[i]= this.allClients.get(i);
        }
        return result;
    }

    @Override
    public void addClient(String s){
        this.allClients.add(s);
    }
}
```

1.3)Server

A lot of things are needed here:

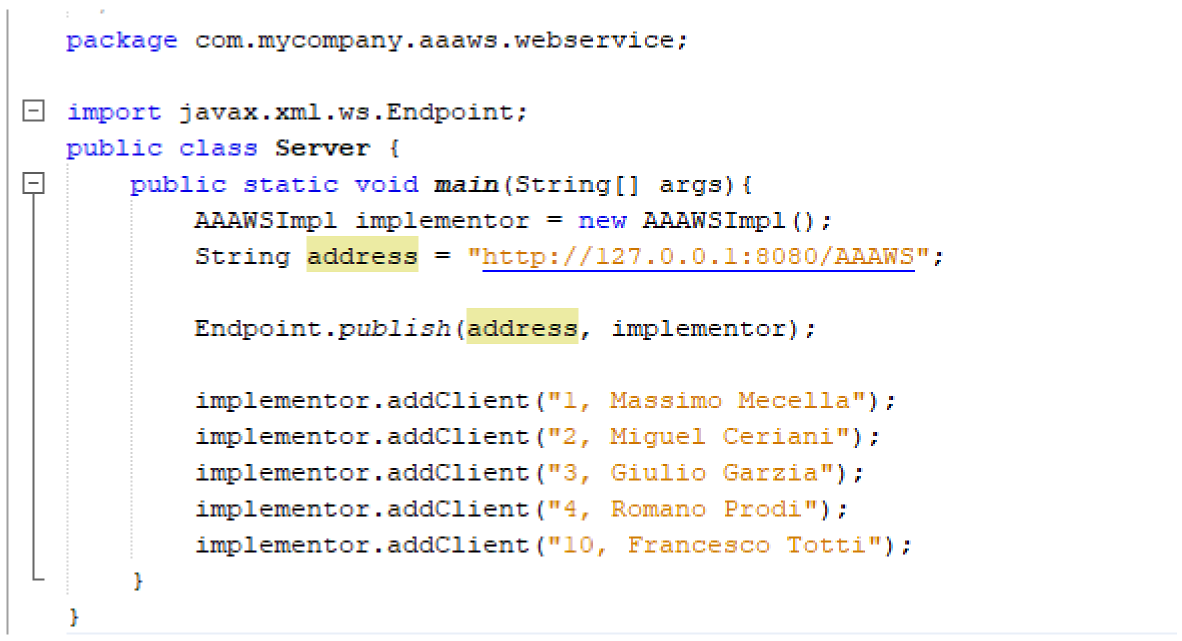
We need:

an **implementor**, that is a AAAWSImpl type

a **String address** to contain the localhost address where our webservice will be

offered, something like "http://localhost:8080/AAAWS"

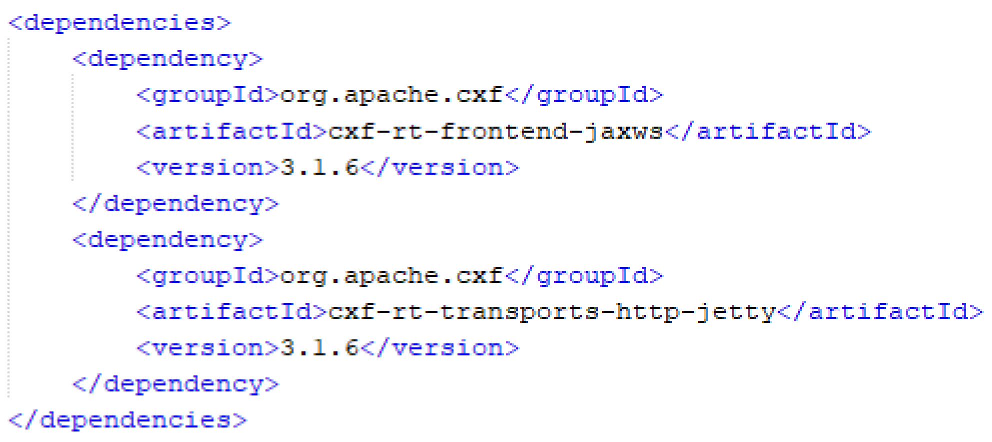
then we publish it by **Endpoint.publish(address, implementor)**



```java
package com.mycompany.aaaws.webservice;

import javax.xml.ws.Endpoint;
public class Server {
    public static void main(String[] args){
        AAAWSImpl implementor = new AAAWSImpl();
        String address = "http://127.0.0.1:8080/AAAWS";

        Endpoint.publish(address, implementor);

        implementor.addClient("1, Massimo Mecella");
        implementor.addClient("2, Miguel Ceriani");
        implementor.addClient("3, Giulio Garzia");
        implementor.addClient("4, Romano Prodi");
        implementor.addClient("10, Francesco Totti");
    }
}
```

Lastly, we need to modify our POM.XML file by adding the needed dependencies:
https://www.baeldung.com/introduction-to-apache-cxf

We create the <dependencies>…</dependencies> and inside we add the needed <dependency>



```xml
<dependencies>
    <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-rt-frontend-jaxws</artifactId>
        <version>3.1.6</version>
    </dependency>
    <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-rt-transports-http-jetty</artifactId>
        <version>3.1.6</version>
    </dependency>
</dependencies>
```

Now, we need to create our **Client** as a new project in the exact same way and a Client.java class

A lot of work have to be done for the client class, let's see them in details.

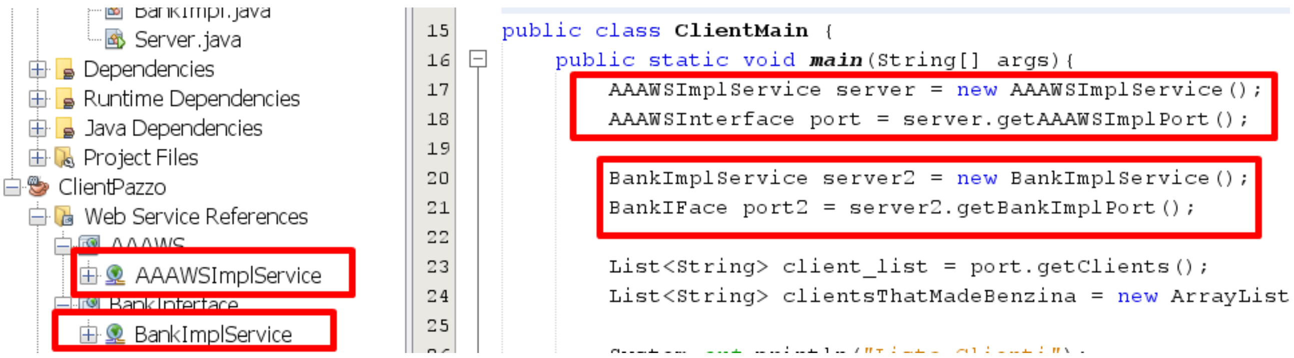
2)Client
The first thing to do, right after the creation of the class is to add all the **webservices** needed, this can be done by:

right click on the project -> new -> Web Service Client

Then on the WSDL URL field we want to add the address we exposed during the call of endpoint.publish, in this case will be localhost:8080/AAAWS**?wsdl.**

This operation has to be repeated for every web-services we want to use.

Every time we add a webservice, a new class will be generated, usually with the name of (methodsImpl+Service), like in this case where AAAWSImpl became **AAAWSImplService**, this class came with a method, **get*****Port()** that will return a interface file (like AAAWSInterface) and from this variable sd



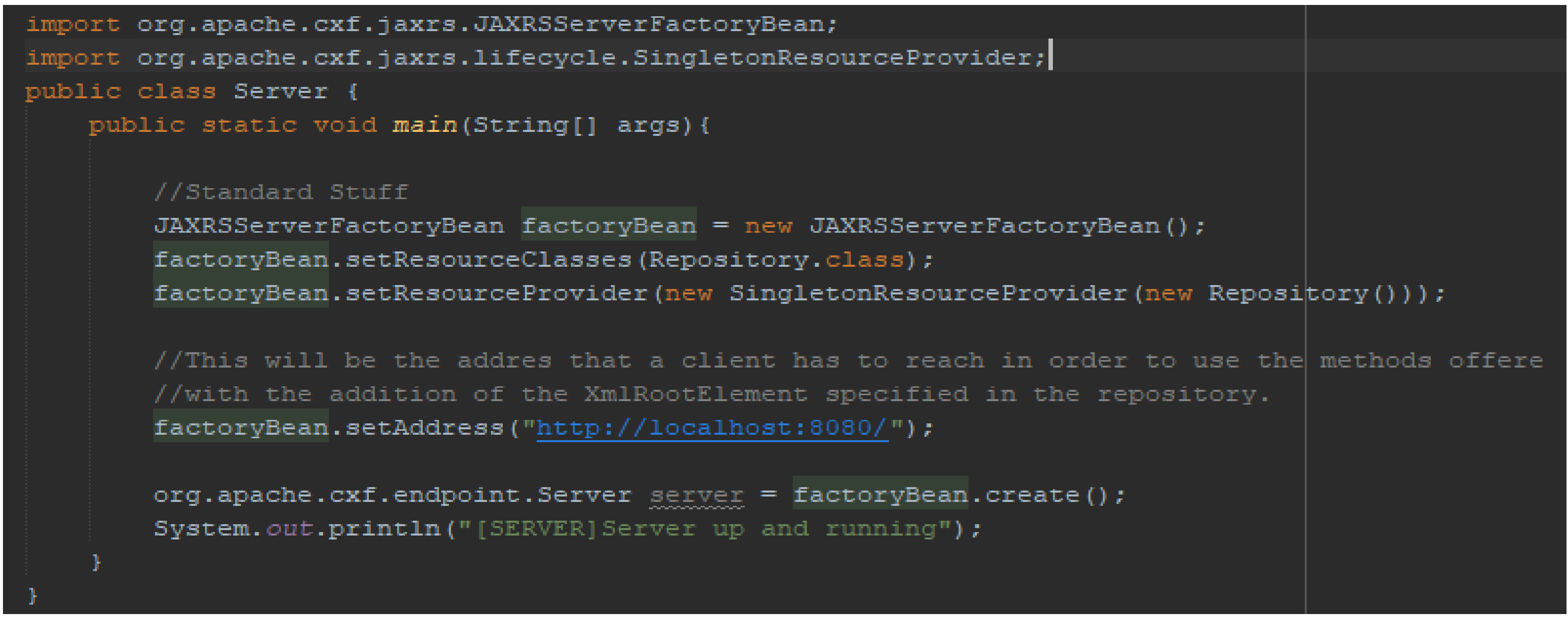
```
public class ClientMain {
    public static void main(String[] args){
        AAAWSImplService server = new AAAWSImplService();
        AAAWSInterface port = server.getAAAWSImplPort();

        BankImplService server2 = new BankImplService();
        BankIFace port2 = server2.getBankImplPort();

        List<String> client_list = port.getClients();
        List<String> clientsThatMadeBenzina = new ArrayList
```

LAB2 – REST Web Services.

1)RESTServer

1.1)Server.java

The main server.java will execute and establish the webserver that host the resourcers users can request.
It is very basic with almost no personalization other than when setting the address.



```java
import org.apache.cxf.jaxrs.JAXRSServerFactoryBean;
import org.apache.cxf.jaxrs.lifecycle.SingletonResourceProvider;
public class Server {
    public static void main(String[] args){

        //Standard Stuff
        JAXRSServerFactoryBean factoryBean = new JAXRSServerFactoryBean();
        factoryBean.setResourceClasses(Repository.class);
        factoryBean.setResourceProvider(new SingletonResourceProvider(new Repository()));

        //This will be the addres that a client has to reach in order to use the methods offere
        //with the addition of the XmlRootElement specified in the repository.
        factoryBean.setAddress("http://localhost:8080/");

        org.apache.cxf.endpoint.Server server = factoryBean.create();
        System.out.println("[SERVER]Server up and running");
    }
}
```

1.2)GenericResources.java

Our server will be in charge of manipulating at least one type of resource, not necessary a String or a basic one, thus it probably need a specific class for it.
A couple of things are important, we have to specify the name of the resource via a:

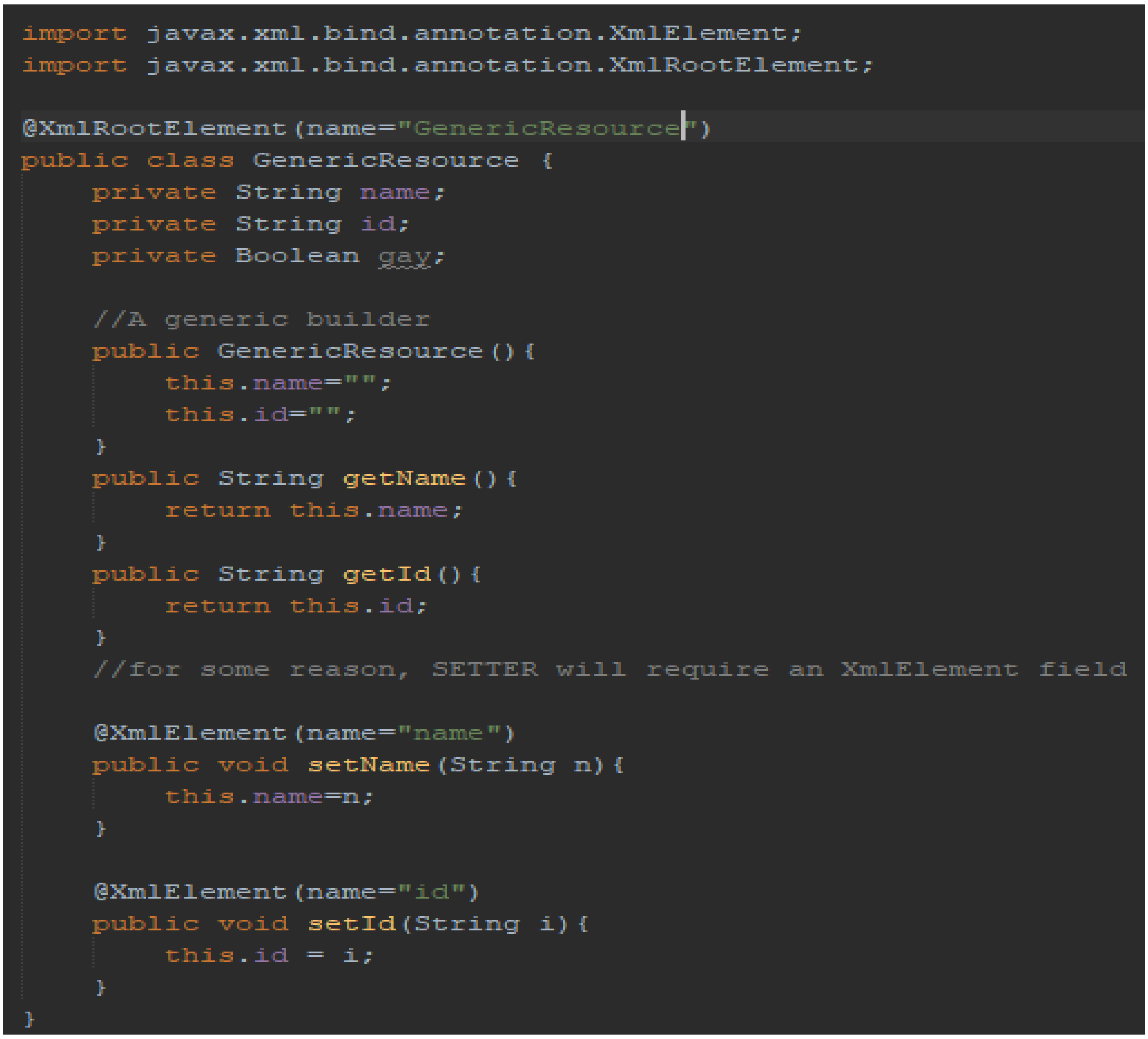
@XmlRootElement(name="GenericResource")

right before the class declaration, then we have to declare ALL the **SETTER** and **GETTER** for all the resourcers, watch out for the name!
String name ➔ setName(..) + getName()

Additionally, all the **SETTER** will require an XML FIELD.

@XmlElement(name="id")

IMPORTANT->This file wil have to be present in both RESTServer and RESTClient packages

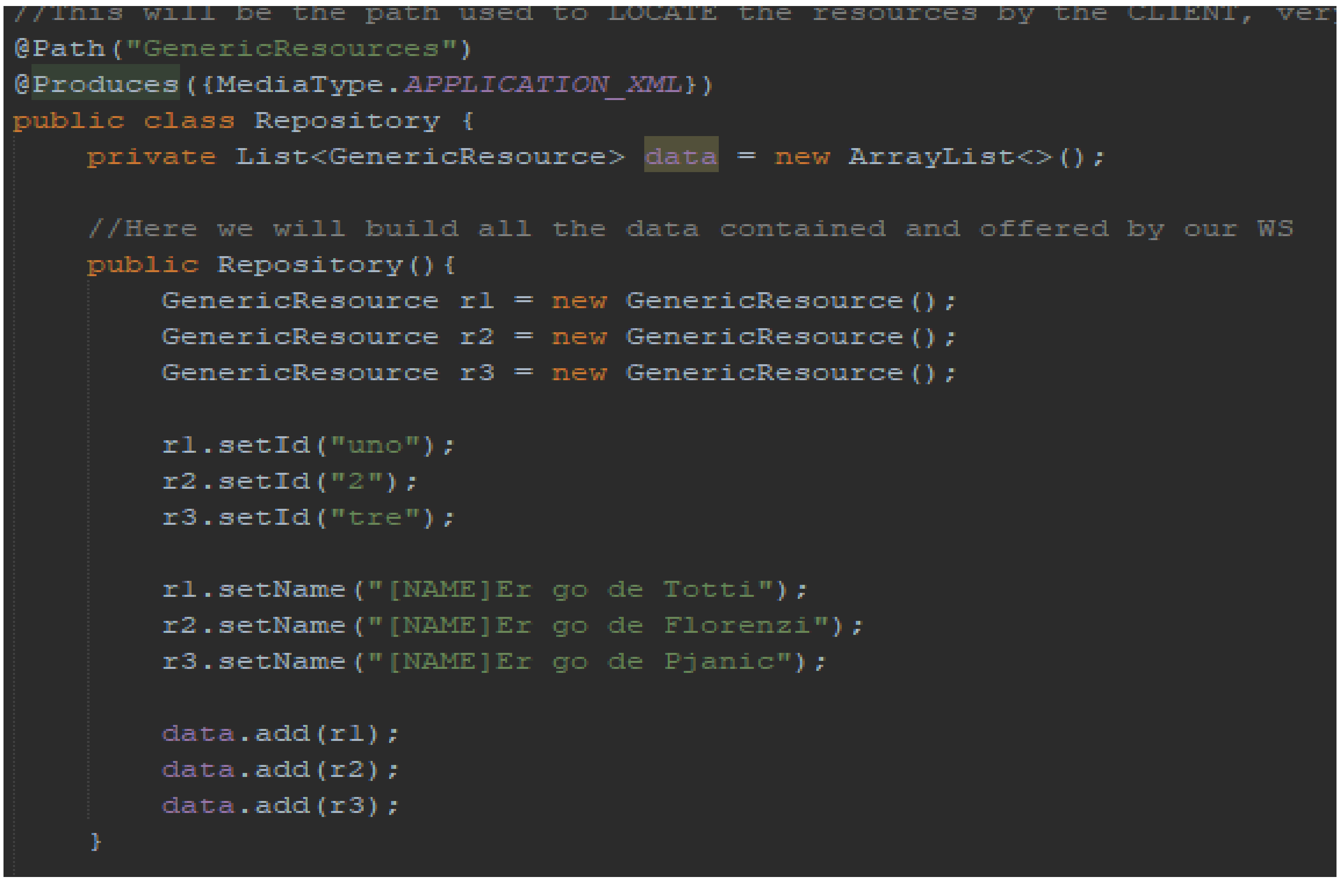


```java
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name="GenericResource")
public class GenericResource {
    private String name;
    private String id;
    private Boolean gay;

    //A generic builder
    public GenericResource(){
        this.name="";
        this.id="";
    }
    public String getName(){
        return this.name;
    }
    public String getId(){
        return this.id;
    }
    //for some reason, SETTER will require an XmlElement field

    @XmlElement(name="name")
    public void setName(String n){
        this.name=n;
    }

    @XmlElement(name="id")
    public void setId(String i){
        this.id = i;
    }
}
```

1.3)Repository.java

Repository file it's much more critical; The first thing to do is to specify the:
@Path("GenericResources"), this is where the CLIENT will have to go in order locate the resources.
@Produces({MediaType.Application_XML}) to specify that we are producing XML-Based file.
^@Produces("text/xml") is equivalent

Then we BUILD UP all the database inside the repository's constructor, those will be all the



```java
//This will be the path used to LOCATE the resources by the CLIENT, ver
@Path("GenericResources")
@Produces({MediaType.APPLICATION_XML})
public class Repository {
    private List<GenericResource> data = new ArrayList<>();

    //Here we will build all the data contained and offered by our WS
    public Repository(){
        GenericResource r1 = new GenericResource();
        GenericResource r2 = new GenericResource();
        GenericResource r3 = new GenericResource();

        r1.setId("uno");
        r2.setId("2");
        r3.setId("tre");

        r1.setName("[NAME]Er go de Totti");
        r2.setName("[NAME]Er go de Florenzi");
        r3.setName("[NAME]Er go de Pjanic");

        data.add(r1);
        data.add(r2);
        data.add(r3);
    }
```

resources we will be able to offer, simply by creating a list of GenericResources

Then, since REST it's an HTTP Based web-service, we will offer access to our methods/function via different HTTP-Request: **GET, POST, PUT, DELETE.**

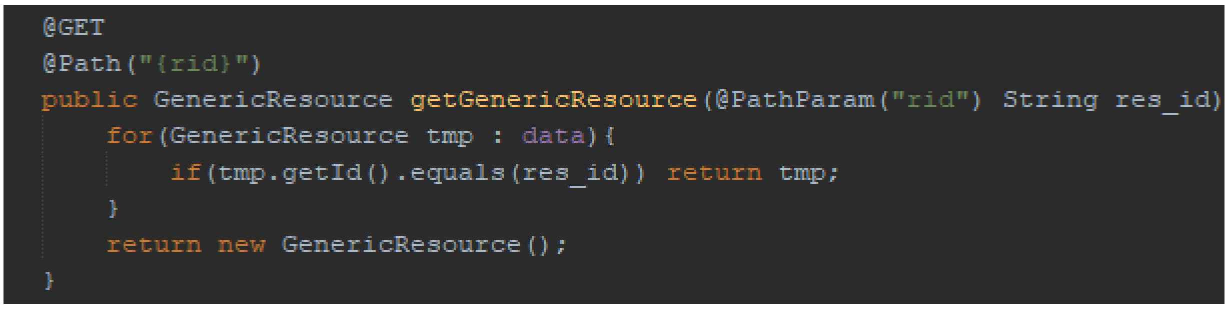
For each methods you want to offer, you will have to specify both the type of http operation:
@GET, @POST, @PUT, @DELETE and a **@Path("{rid}")** when the method require a param inside the addr.

Then for each methods we can return both an object (return new GenricResource()), ora a particular type of HTTP Response (return Response)

SERVER GET With Param: As specified before, we need:
@GET
@Path("{rid}")

Also, this get request will always return a GenericResource since it is a GET operation.

->public GenericResource getGenericResource**(@PathParam("rid") String id){ … }**



```
@GET
@Path("{rid}")
public GenericResource getGenericResource(@PathParam("rid") String res_id)
    for(GenericResource tmp : data){
        if(tmp.getId().equals(res_id)) return tmp;
    }
    return new GenericResource();
}
```

Here we use @PathParam to specify that we are receving a String id as param in the HTTP request.

//

SERVER SECONDARY GET: You may want to have multiple GET operation, maybe performing different tasks. What you want to change is the PATH.
Assuming you already have the GET Request above…
@GET
@Path("/capitani")
public GenericResource getCapitani(){

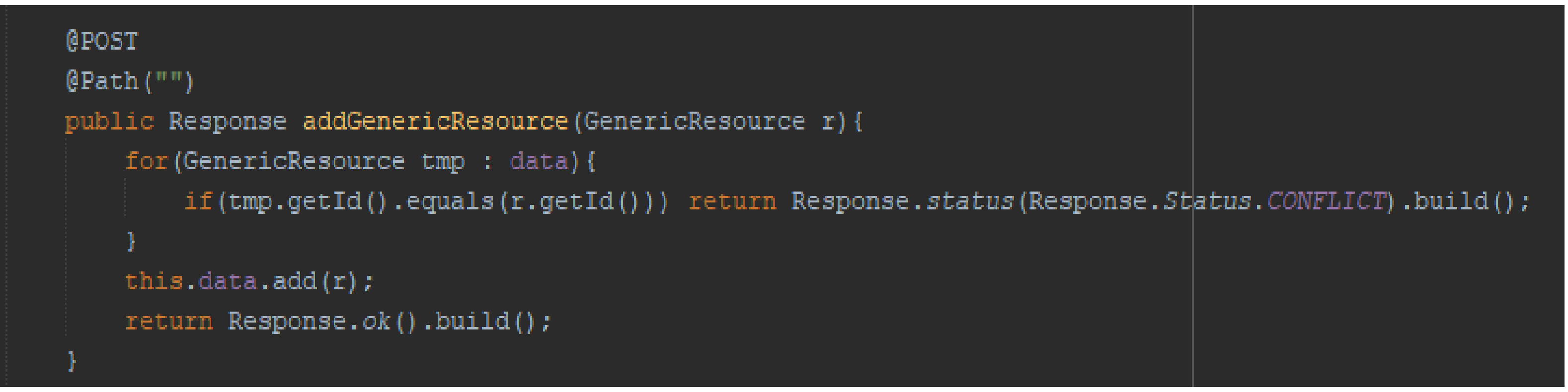
Now the path to access this methods will no longer be the one at the beginning of the Repository (@Path("GenericResources")), but instead it will be:
/GenericResources/capitani

//

SERVER POST: This is a post request, it can only return different type of RESPONSE since it is used to send and ask the addition of some data to ones stored by the server.
@POST
@Path("") // Note how nothing is passed as http parameter, but instead we can directly provide the resource we need directly as a parameter for the request.

-> public Response addGenericResource(**GenericResource** r)
…
return Response.status(Response.Status.CONFLICT).build();
return Response.ok().build();

}



```java
@POST
@Path("")
public Response addGenericResource(GenericResource r){
    for(GenericResource tmp : data){
        if(tmp.getId().equals(r.getId())) return Response.status(Response.Status.CONFLICT).build();
    }
    this.data.add(r);
    return Response.ok().build();
}
```

//

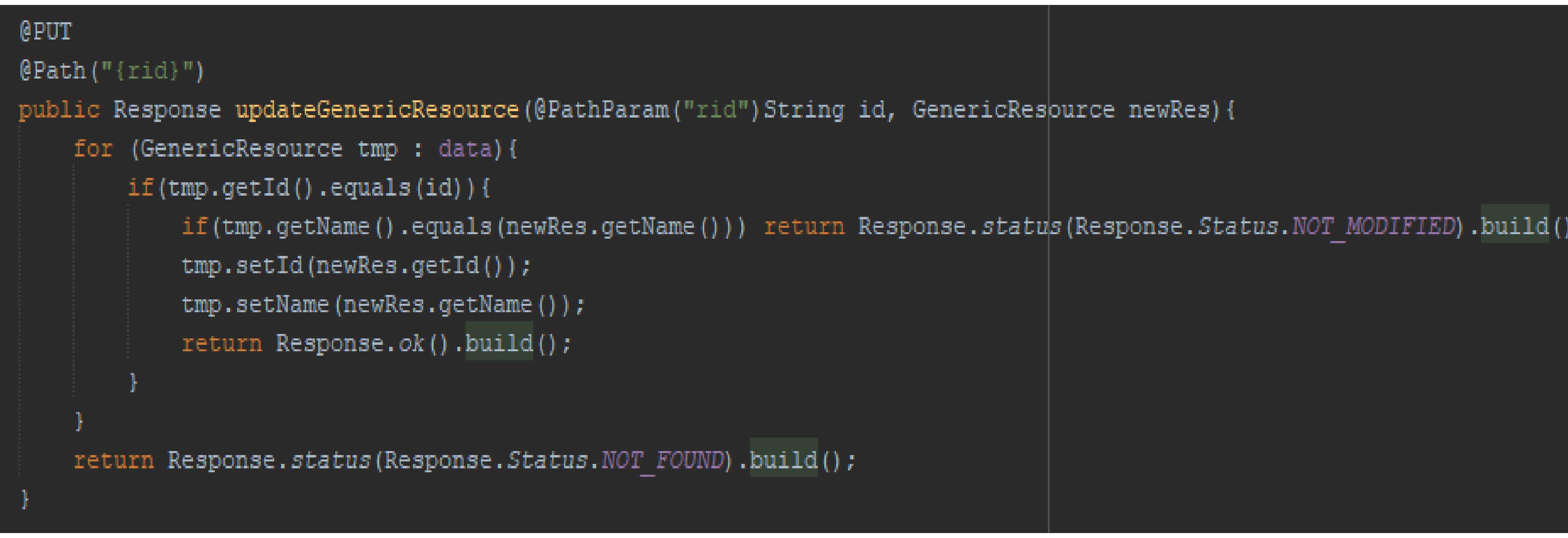
SERVER PUT: Put is similar to post, it is used to MODIFY the value of a value already stored in the database,
this time we will use both a variable/param passed in the HTTP Request (like in the GET), and also a second one passed directly as argument (like in the post).
@PUT
@Path("{rid}")

public Response updateGenericResource**(@PathParam("rid")String id,**
GenericResource newRes){

…
return Response.status(Response.Status.NOT_MODIFIED).build();

return Response.ok().build();

return Response.status(Response.Status.NOT_FOUND).build();}



```java
@PUT
@Path("{rid}")
public Response updateGenericResource(@PathParam("rid")String id, GenericResource newRes){
    for (GenericResource tmp : data){
        if(tmp.getId().equals(id)){
            if(tmp.getName().equals(newRes.getName())) return Response.status(Response.Status.NOT_MODIFIED).build(
            tmp.setId(newRes.getId());
            tmp.setName(newRes.getName());
            return Response.ok().build();
        }
    }
    return Response.status(Response.Status.NOT_FOUND).build();
}
```

//

SERVER DELETE: Used to remove an entry (if present) from the dataset

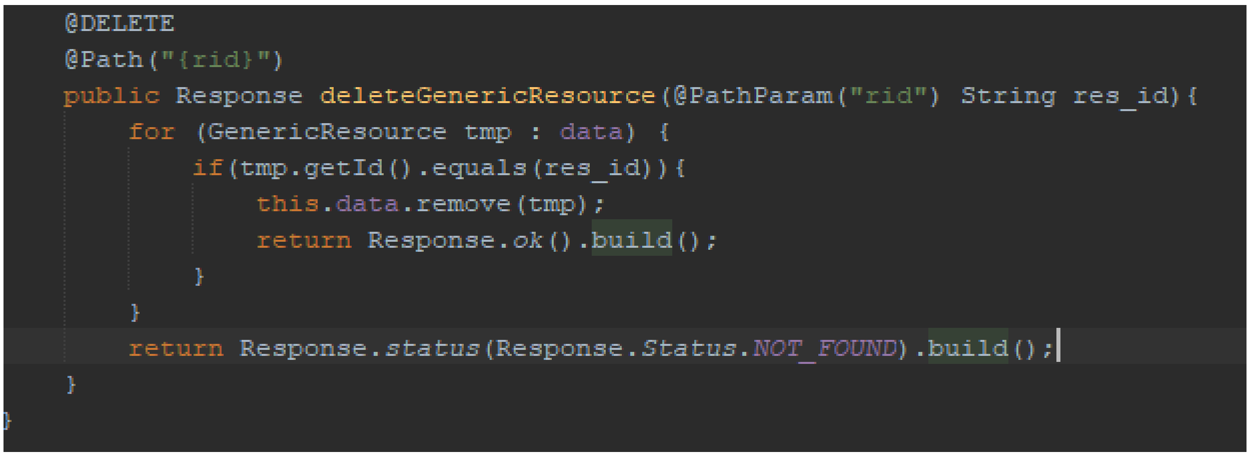
@DELETE
@Path("{rid}")

public Response deleteGenericResource(**@PathParam("rid") String res_id**){

…
return Response.ok().build();

return Response.status(Response.Status.NOT_FOUND).build();

}



```
    @DELETE
    @Path("{rid}")
    public Response deleteGenericResource(@PathParam("rid") String res_id){
        for (GenericResource tmp : data) {
            if(tmp.getId().equals(res_id)){
                this.data.remove(tmp);
                return Response.ok().build();
            }
        }
        return Response.status(Response.Status.NOT_FOUND).build();
    }
}
```

//

In gerenal the way those kind of HTTP-RESPONSE are generate it's always very similar.

For SUCCESS CONFIRMATION: you simply have a generic: **Response.ok().build()**

For every other types of error, hou have a:

Response.status(Response.Status.TYPE_OF_ERROR).build();

Note how every response always and with a **.buid()**.

2)RESTClient

2.1)GenericResources: Clone of 1.2

2.2)Client:

A lot of preparation is needed in the client,

1) first we have to provide what we can see as the
BASE_URL, it will be the one specified in the SERVER file with the addition of the PATH inside the REPOSITORY FILE, so “http://localhost:8080/" + “GenericResources” + “/”

BASE_URL= http://localhost:8080/GenericResources/

2) Then we have to create a CLIENT variable as a **private static CloseableHttpClient client**
That will be used by 2 MANDATORY call, the one to CREATE a client:

```
client = HttpClients.createDefault();
```

And the one to CLOSE it:

```
client.close();
```

//

CLIENT GET OPERATION: As we alreay told in the GET explanation in the server
, we are providing an IDENTIFIER inside our http request to specify the object we are looking for, for that reason the
URI of the resource we are looking for will be **BASE_URL + resourceID**

->http://localhost:8080/GenericResources/uno

We have to create:

```
HttpGet httpGet = new HttpGet(BASE_URL + resID);
httpGet.setHeader("Content-Type","text/xml")
```

Then we send it with

```
HttpResponse response = client.execute(httpGet);
```

Finally we obtain the result with:

```
GenericResource received = JAXB.unmarshal(
        response.getEntity().getContent(), GenericResource.class);
```

//

CLIENT SECONDARY GET OPERATION: Pretty much identic to the normal get, but this time we have to add the additional directory specified in the repository

CLIENT POST OPERATION: This time we are not passing an identifier in the URL but instead an entire object (GeneriResoruce) with the HTTP-Post request.

Also this time we create:

```
HttpPost httpPost = new HttpPost(BASE_URL); //path of POST is empty.
httpPost.setHeader("Content-Type","text/xml");

GenericResource toAdd = new GenericResource()
toAdd.setId(id);
toAdd.setName(name); // The resource we want to add.
```

Then some dark magic stuff to create the packet itself.

```
JAXBContext jaxbContext = JAXBContext.newInstance(GenericResource.class);
Marshaller jaxbMarshaller = jaxbContext.createMarshaller();
jaxbMarshaller.marshal(toAdd, new File("res.xml"));
File file = new File("res.xml");
InputStream targetStream = new FileInputStream(file);
httpPost.setEntity(new InputStreamEntity(targetStream));
```

Then finally we can send it.

```
HttpResponse response=client.execute(httpPost);
```

//

CLIENT PUT OPERATION: We are passing bot a value in the HTTP-Request inside the URI, and both a complete param, it's the union of the requests above.

This time we have to create:

```
HttpPut httpPut = new HttpPut(BASE_URL+oldID);
httpPut.setHeader("Content-Type","text/xml");

GenericResource replacement = new GenericResource();
replacement.setId(id);
replacement.setName(name);
```

Then some dark magic stuff to create the packet itself.

```
JAXBContext jaxbContext = JAXBContext.newInstance(GenericResource.class)
Marshaller jaxbMarshaller = jaxbContext.createMarshaller()
jaxbMarshaller.marshal(replacement, new File("res.xml"));
File file = new File("res.xml");
InputStream targetStream = new FileInputStream(file);
httpPost.setEntity(new InputStreamEntity(targetStream));
```

Then finally we can send it.

```
        HttpResponse response=client.execute(httpPut);
```

///

CLIENT DELETE OPERATION: We are deleting an item indicated as param in the http request, so i twill be very similar to the GET:

```
HttpDelete httpDelete = new HttpDelete(BASE_URL+id)
httpDelete.setHeader("Content-Type","text/xml"
HttpResponse response = client.Execute(httpDelete)
```

///

LAB3 – JMS

The idea of JMS is to create TOPIC where PUBLISHER can send message (that will be reached by any subscriber) and SUBSCRIBER can subscribe to a given TOPIC and receive only the messages they are interest into.

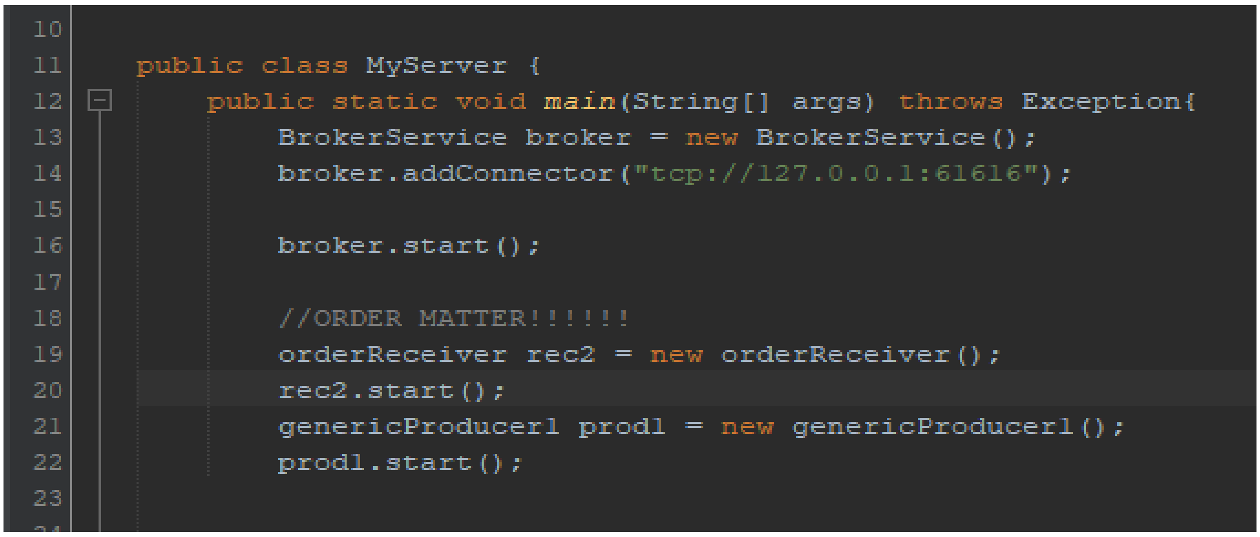
In order to **send** message in a topic (like genericProducer) you need a **Destination**, a **MessageProducer** and a Text **Message**.

In order to **receive** you need (like in the client) you need a **Destination**, a **MessageConsumer** and a **MessageListener** associated to that MessageConsumer.

1)RESTServer

1.1 MyServer

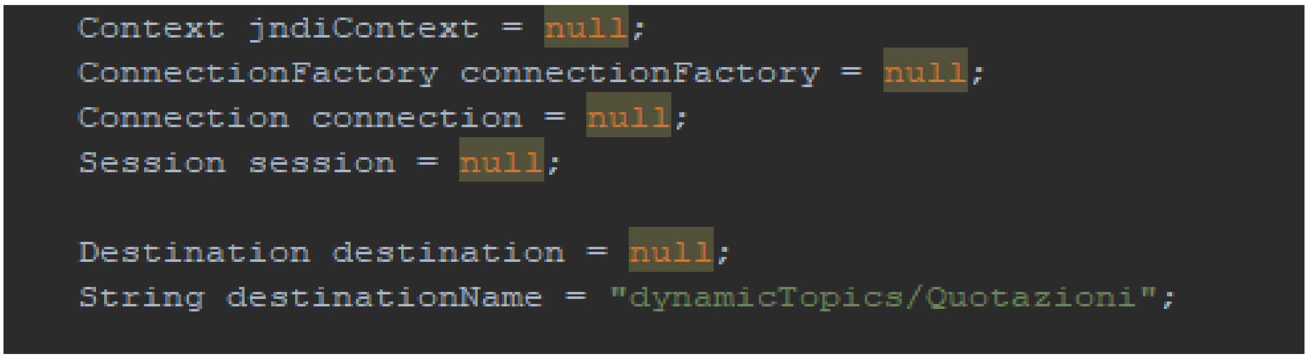
Very simple, we need to create a BROKER to lock a given TCP:ADDRESS:PORT, like
-> **tcp://127.0.0.1:61616**, and them start all the produer/receiver that we want, in this case we will need 2 manager.



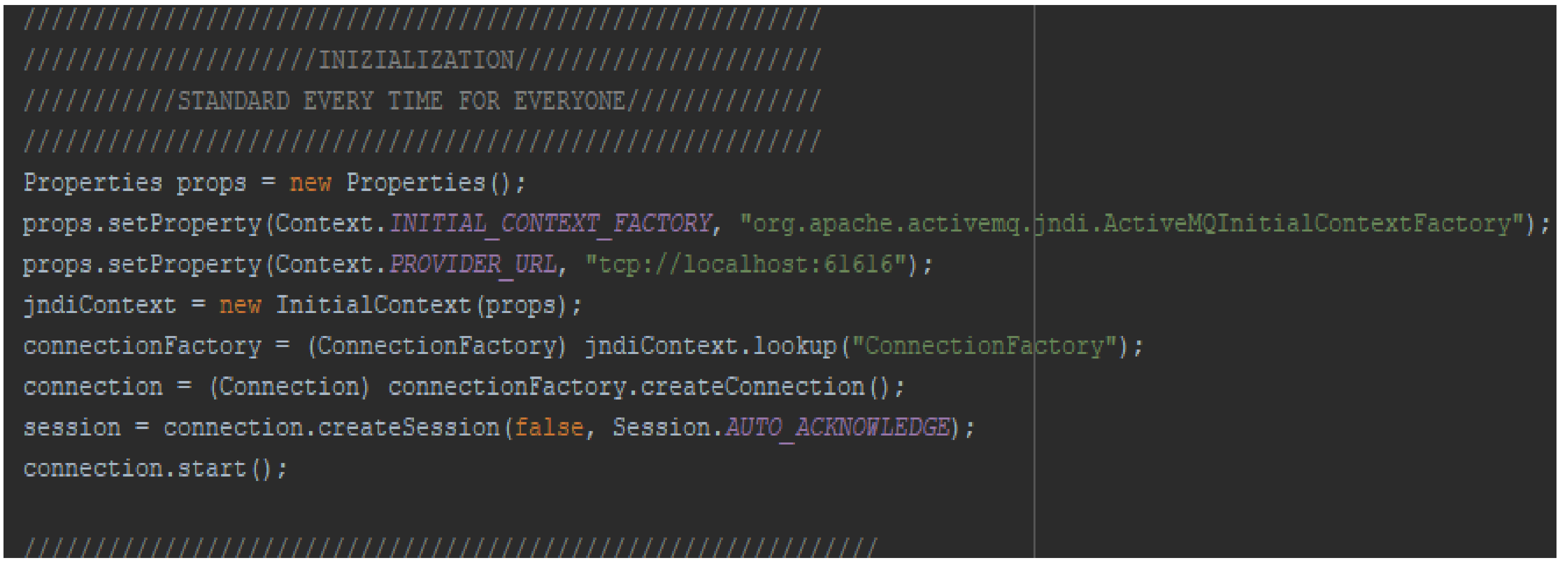
```java
    public class MyServer {
        public static void main(String[] args) throws Exception{
            BrokerService broker = new BrokerService();
            broker.addConnector("tcp://127.0.0.1:61616");

            broker.start();

            //ORDER MATTER!!!!!!
            orderReceiver rec2 = new orderReceiver();
            rec2.start();
            genericProducer1 prod1 = new genericProducer1();
            prod1.start();

```

1.2 genericProducer

A **LOT** of inizialization settings are shared between all the server and all the clients and they are almost identical.
The last one is the name of the topic we will refer to, in this case to WRITE.

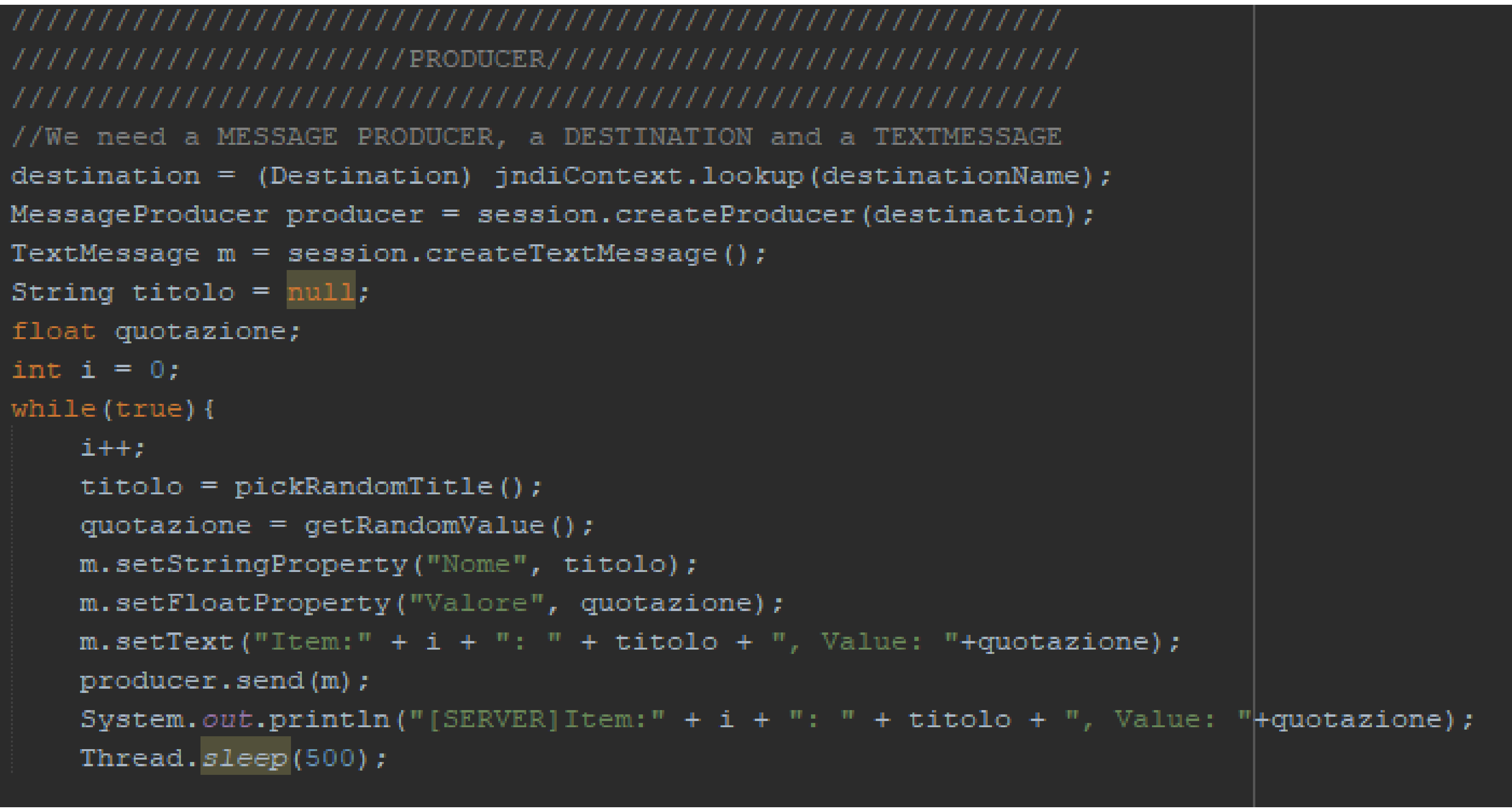


```java
    Context jndiContext = null;
    ConnectionFactory connectionFactory = null;
    Connection connection = null;
    Session session = null;

    Destination destination = null;
    String destinationName = "dynamicTopics/Quotazioni";
```



```
////////////////////////////////////////////////////////////
/////////////////////INIZIALIZATION//////////////////////
///////////STANDARD EVERY TIME FOR EVERYONE//////////////
////////////////////////////////////////////////////////////
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);
connectionFactory = (ConnectionFactory) jndiContext.lookup("ConnectionFactory");
connection = (Connection) connectionFactory.createConnection();
session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
connection.start();

////////////////////////////////////////////////////////////////
```

To send data in this topic we need:
a **MessageProducer** as **producer**
a **Destination** created with the **destinationName** (The topic's name)
and **TextMessge** to be sent under the form of **Message.**



```
//////////////////////////////////////////////////////////////
///////////////////////PRODUCER/////////////////////////////////
//////////////////////////////////////////////////////////////
//We need a MESSAGE PRODUCER, a DESTINATION and a TEXTMESSAGE
destination = (Destination) jndiContext.lookup(destinationName);
MessageProducer producer = session.createProducer(destination);
TextMessage m = session.createTextMessage();
String titolo = null;
float quotazione;
int i = 0;
while(true){
    i++;
    titolo = pickRandomTitle();
    quotazione = getRandomValue();
    m.setStringProperty("Nome", titolo);
    m.setFloatProperty("Valore", quotazione);
    m.setText("Item:" + i + ": " + titolo + ", Value: "+quotazione);
    producer.send(m);
    System.out.println("[SERVER]Item:" + i + ": " + titolo + ", Value: "+quotazione);
    Thread.sleep(500);
```

Note that once we have a TextMessage we can add and populate its field by using command like:
m.**setStringProperty("Name of The prop", String)**
m.**setFloatProperty("Name of the prof", float).**
m.**setText(String)"**

Note that the NAME_OF_THE_PROP is important when the SUBSCRIBER/CONSUMER will want to access the TextMessage.

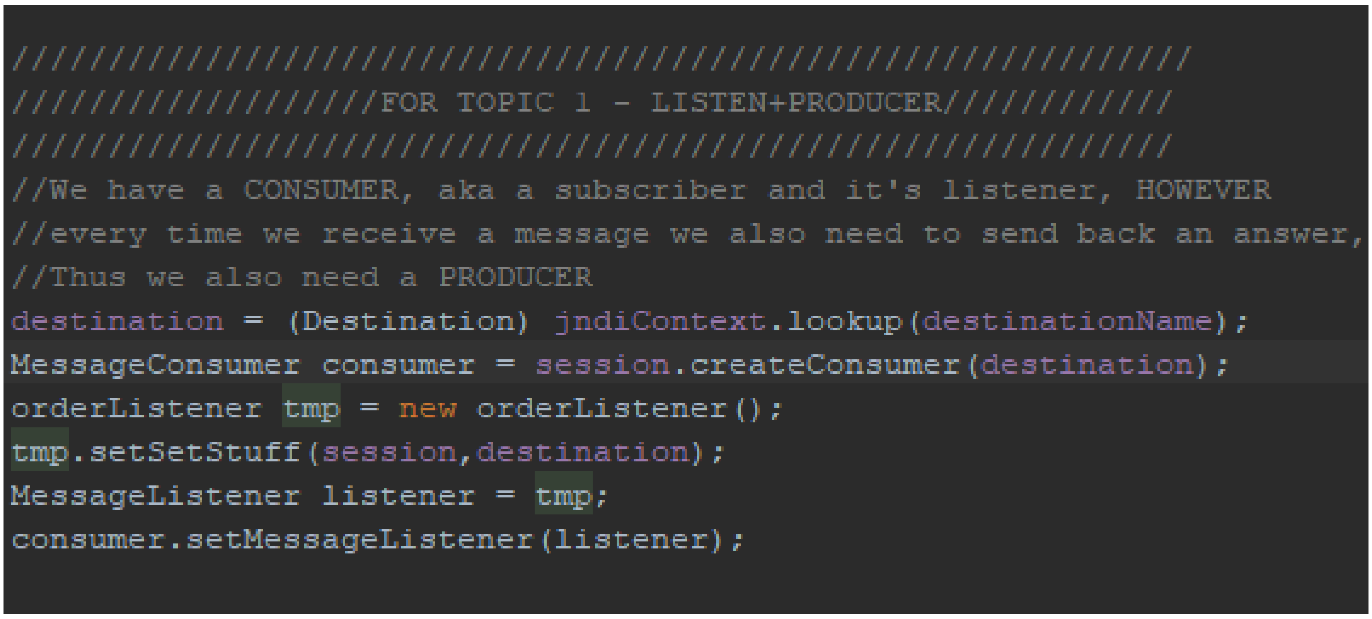
Then finally we can send it with a **producer.send(m)**

1.3 orderReceiver:

Our server have a functionality that not only SEND stuff via our GENERICPRODUCER (1.2) but also receive another type of messages in another topic!

Funny fact, the receiving part is exactly the same of the Client that we will se more in detail later, long story short, you need a **Destination,** a **MessageConsumer** and a **MessageListener** associated to that message consumer.

To spicy it up, we want that to answer to that message, for that reason will open a **PRODUCER** INSIDE the messageListener, to do so we send both a **SESSION** and a **DESTINTION** to it, since they are exactly the same we are using also to receive messages.



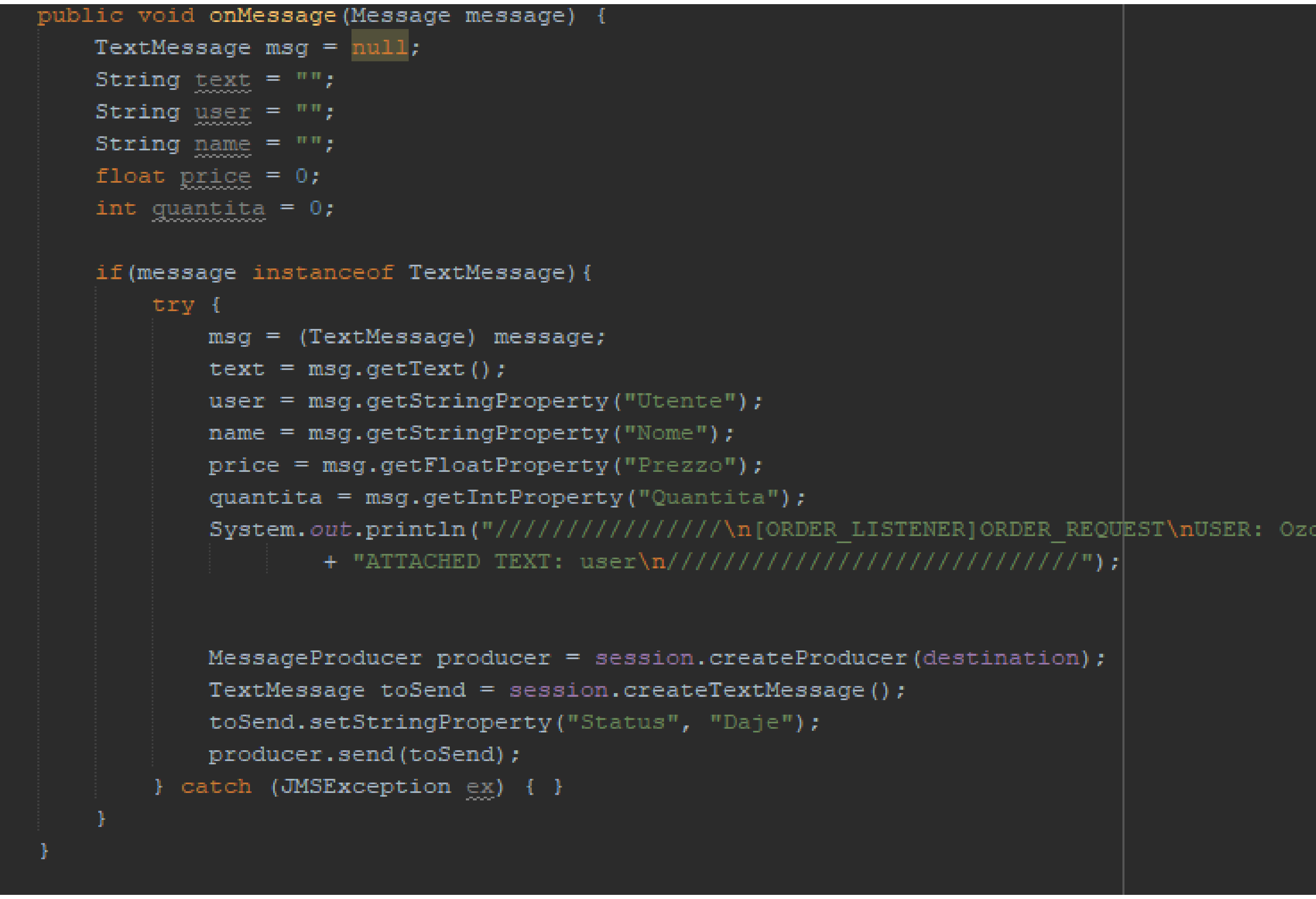
```
////////////////////////////////////////////////////////////
///////////////////FOR TOPIC 1 - LISTEN+PRODUCER///////////
///////////////////////////////////////////////////////////
//We have a CONSUMER, aka a subscriber and it's listener, HOWEVER
//every time we receive a message we also need to send back an answer,
//Thus we also need a PRODUCER
destination = (Destination) jndiContext.lookup(destinationName);
MessageConsumer consumer = session.createConsumer(destination);
orderListener tmp = new orderListener();
tmp.setSetStuff(session,destination);
MessageListener listener = tmp;
consumer.setMessageListener(listener);
```

1.4 orderListener

We will see listener more in detail later, but the idea is that each time we create a MESSAGELISTENER we are IMPLEMENTING it and thus we have to implement its method, that is **onMessage**, triggered every time a message on that topic is received.
The pattern to manage the the listener is always the same.

We check if the Message we received is **instanceof** TextMessage, in that case we can cast it and retreive all of its field (the one we added with m.setFloatProperty/m.setStringpProperty) with a similar command.
m.getStringProperty(Name_Of_The_Property);

Then in this case we used the session and the destination to create a MessageProducer, then we created our TextMessage and we sended it.

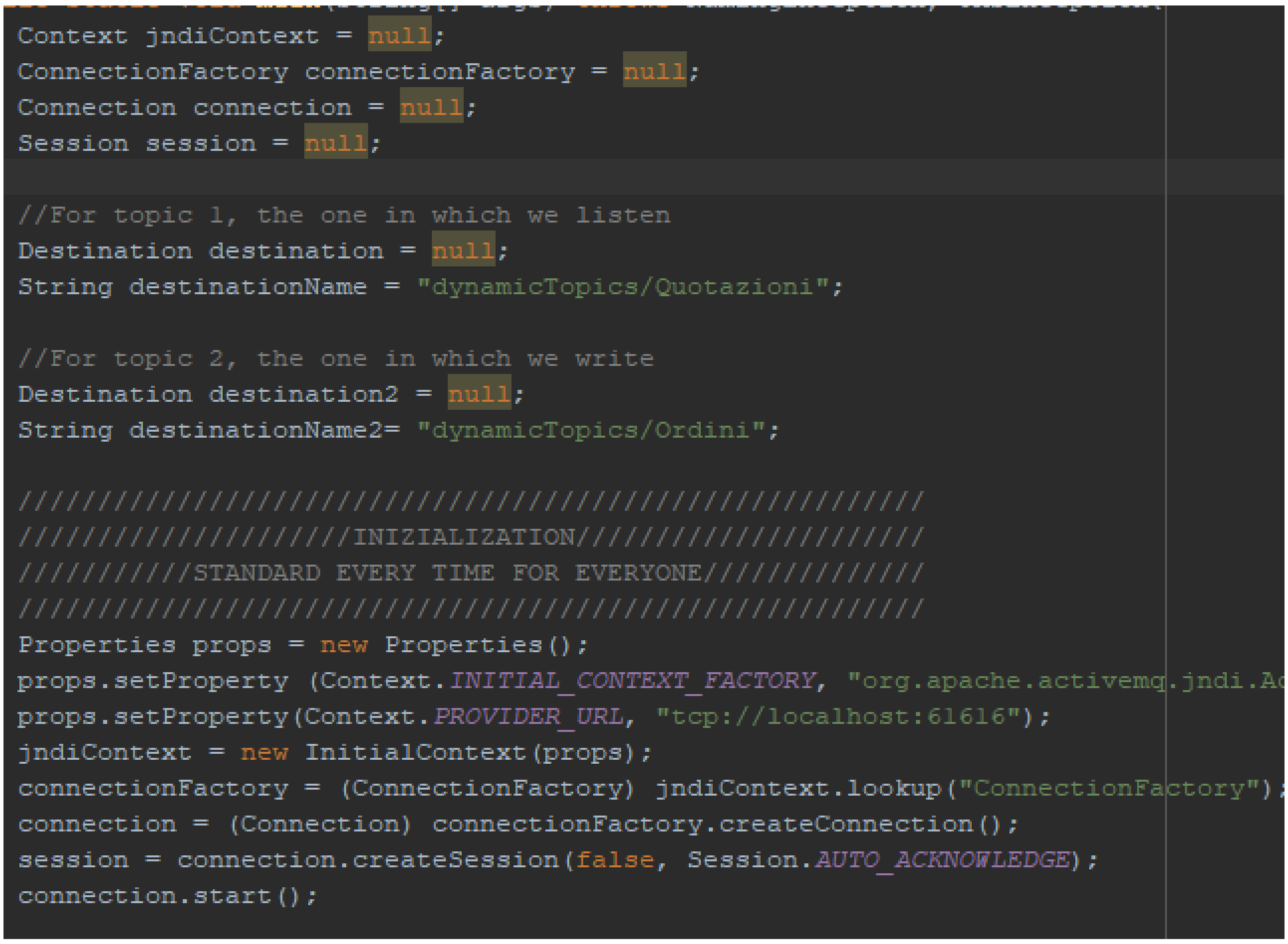


```java
public void onMessage(Message message) {
    TextMessage msg = null;
    String text = "";
    String user = "";
    String name = "";
    float price = 0;
    int quantita = 0;

    if(message instanceof TextMessage){
        try {
            msg = (TextMessage) message;
            text = msg.getText();
            user = msg.getStringProperty("Utente");
            name = msg.getStringProperty("Nome");
            price = msg.getFloatProperty("Prezzo");
            quantita = msg.getIntProperty("Quantita");
            System.out.println("////////////////\n[ORDER_LISTENER]ORDER_REQUEST\nUSER: Ozo
                    + "ATTACHED TEXT: user\n/////////////////////////////");


            MessageProducer producer = session.createProducer(destination);
            TextMessage toSend = session.createTextMessage();
            toSend.setStringProperty("Status", "Daje");
            producer.send(toSend);
        } catch (JMSException ex) { }
    }
}
```

2)RESTClient

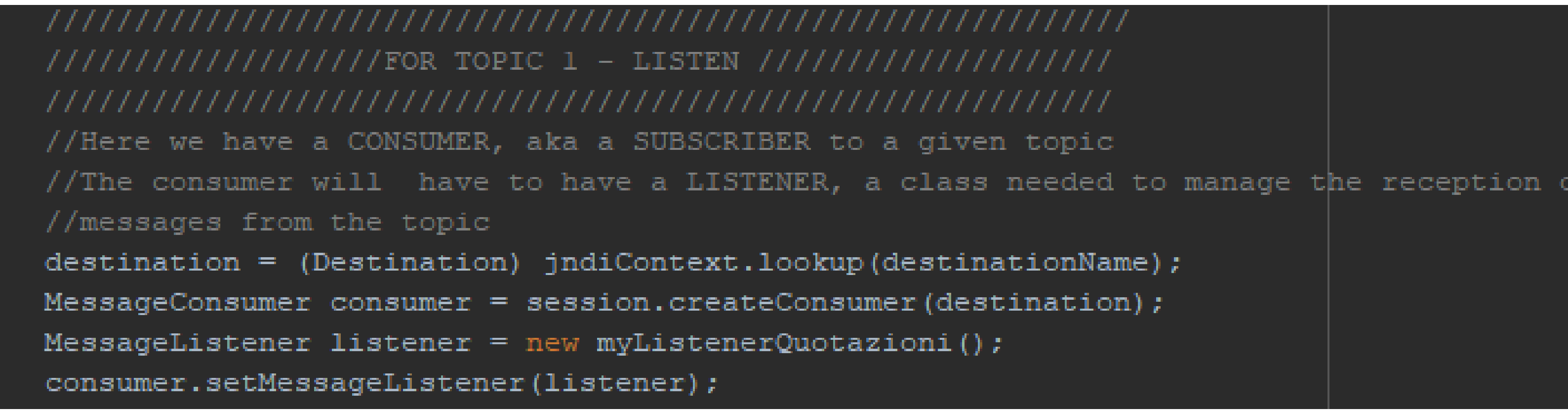
1.1 MyClient

Very simple, same start as usual (we have two topic at the same time but ther's no difference).



```
Context jndiContext = null;
ConnectionFactory connectionFactory = null;
Connection connection = null;
Session session = null;

//For topic 1, the one in which we listen
Destination destination = null;
String destinationName = "dynamicTopics/Quotazioni";

//For topic 2, the one in which we write
Destination destination2 = null;
String destinationName2= "dynamicTopics/Ordini";

/////////////////////////////////////////////////////////
/////////////////////INIZIALIZATION/////////////////////
///////////STANDARD EVERY TIME FOR EVERYONE/////////////
/////////////////////////////////////////////////////////
Properties props = new Properties();
props.setProperty (Context.INITIAL_CONTEXT_FACTORY, "org.apache.activemq.jndi.Ac
props.setProperty(Context.PROVIDER_URL, "tcp://localhost:61616");
jndiContext = new InitialContext(props);
connectionFactory = (ConnectionFactory) jndiContext.lookup("ConnectionFactory");
connection = (Connection) connectionFactory.createConnection();
session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
connection.start();
```

For the first TOPIC we only want to receive update, so we need to create a **DESTINATION** with the **destination name** (the topic's name), a **Message Consumer** and a **Message Listener** associated to that consumer.



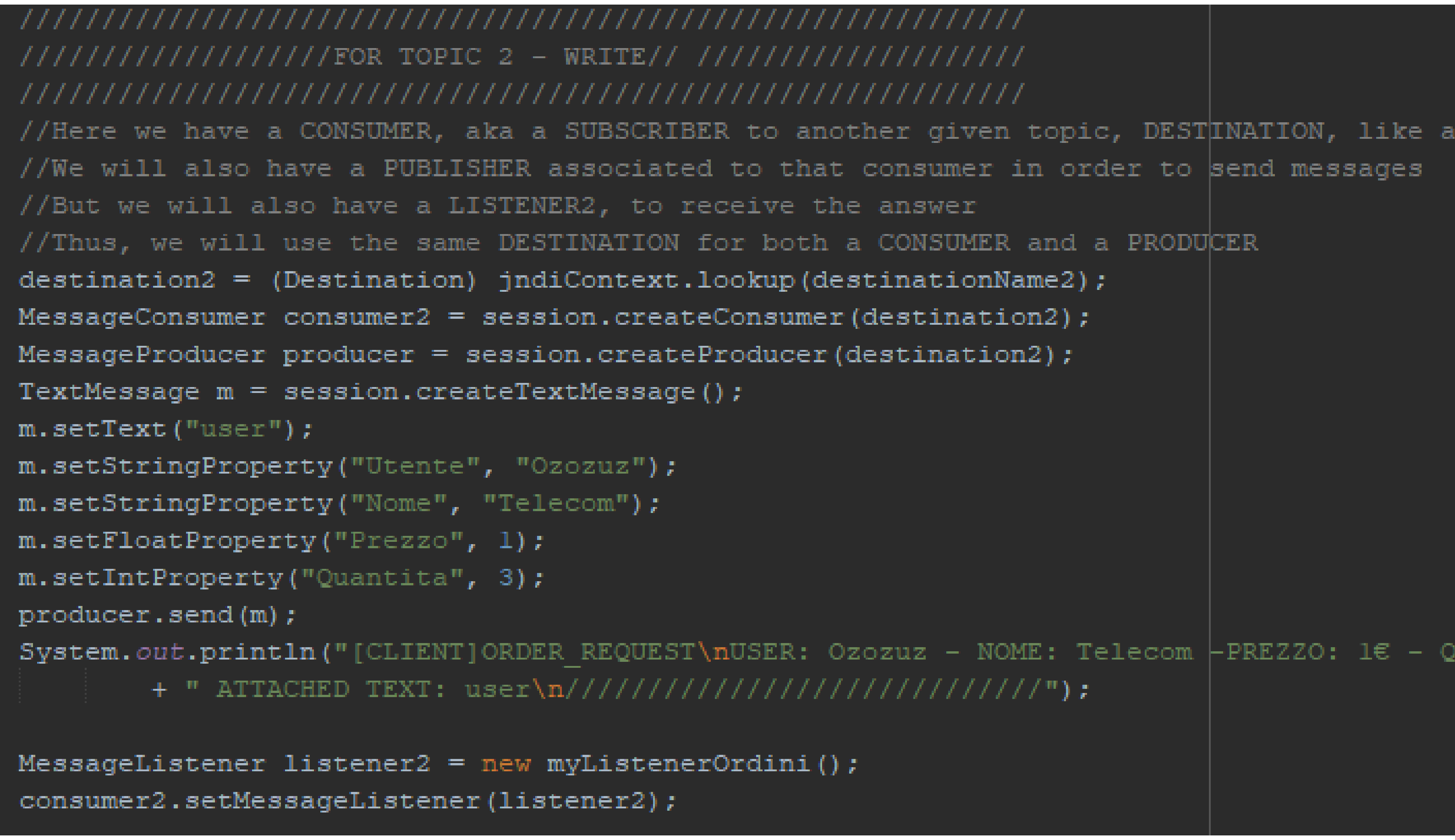
```
///////////////////////////////////////////////////////////////
////////////////////FOR TOPIC 1 - LISTEN /////////////////////
//////////////////////////////////////////////////////////////
//Here we have a CONSUMER, aka a SUBSCRIBER to a given topic
//The consumer will  have to have a LISTENER, a class needed to manage the reception o
//messages from the topic
destination = (Destination) jndiContext.lookup(destinationName);
MessageConsumer consumer = session.createConsumer(destination);
MessageListener listener = new myListenerQuotazioni();
consumer.setMessageListener(listener);
```

For the second topic instead we want first to send a message in that topic, and then we want to listen for the incoming answer.

For that reason we need:

DESTINATION (For both)

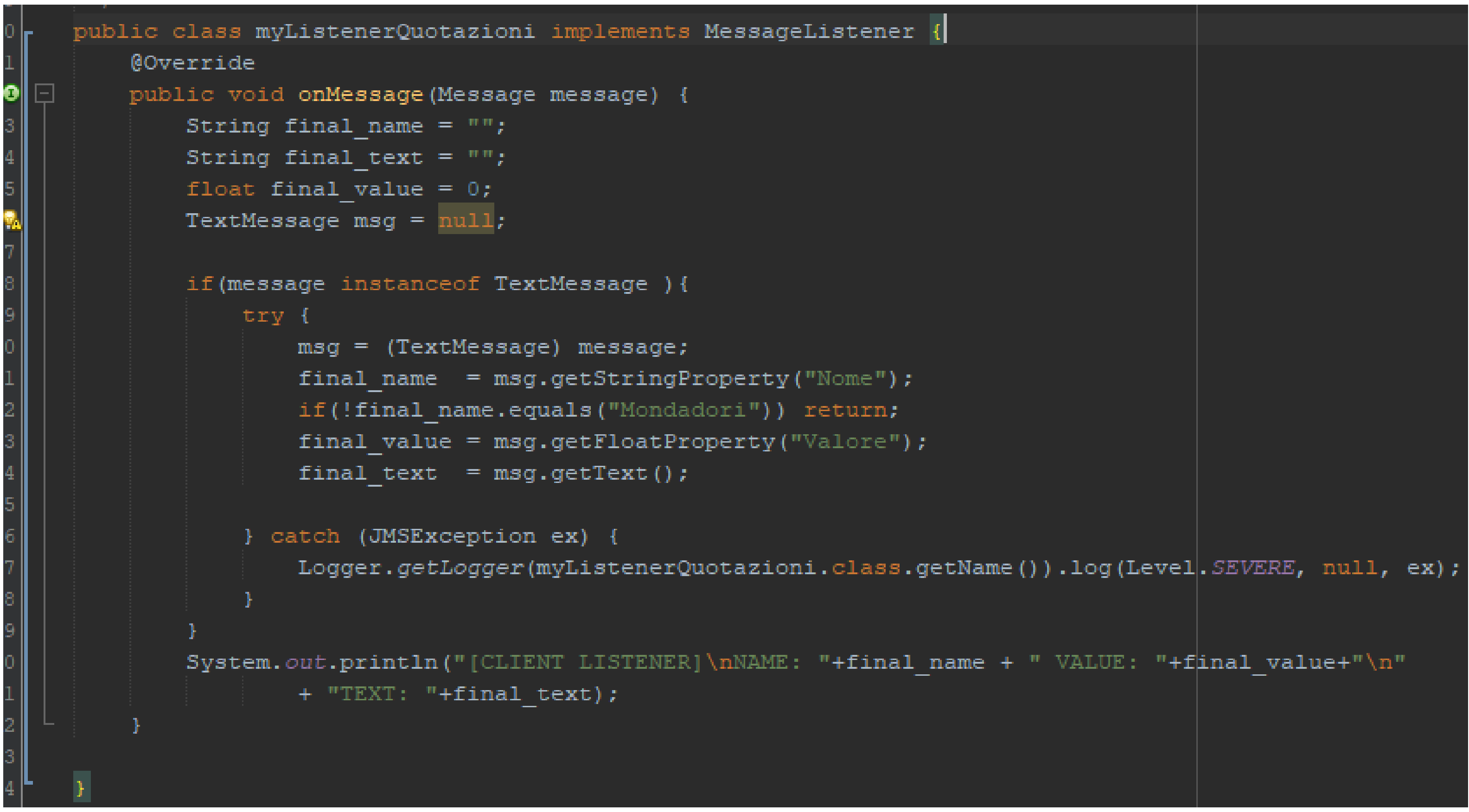
MESSAGE CONSUMER	**MESSAGE PRODUCER**
MESSAGE LISTENER	**TEXTMESSAGE**



```
/////////////////////////////////////////////////////////////
///////////////////FOR TOPIC 2 - WRITE// ////////////////////
/////////////////////////////////////////////////////////////
//Here we have a CONSUMER, aka a SUBSCRIBER to another given topic, DESTINATION, like a
//We will also have a PUBLISHER associated to that consumer in order to send messages
//But we will also have a LISTENER2, to receive the answer
//Thus, we will use the same DESTINATION for both a CONSUMER and a PRODUCER
destination2 = (Destination) jndiContext.lookup(destinationName2);
MessageConsumer consumer2 = session.createConsumer(destination2);
MessageProducer producer = session.createProducer(destination2);
TextMessage m = session.createTextMessage();
m.setText("user");
m.setStringProperty("Utente", "Ozozuz");
m.setStringProperty("Nome", "Telecom");
m.setFloatProperty("Prezzo", 1);
m.setIntProperty("Quantita", 3);
producer.send(m);
System.out.println("[CLIENT]ORDER_REQUEST\nUSER: Ozozuz - NOME: Telecom -PREZZO: 1€ - Q
        + " ATTACHED TEXT: user\n/////////////////////////////");

MessageListener listener2 = new myListenerOrdini();
consumer2.setMessageListener(listener2);
```

1.2 MyListener

As explained before, MyListener will be an implementation of MessageListener,
Steps are simple, we check if the Message received is an **instanceof** TextMessage, in that case we cast it to a local variable of us and we use the latter to retreive alle the value and attributes using operation like m.**getStrinProperty/getText/getFloatProperty** etc.

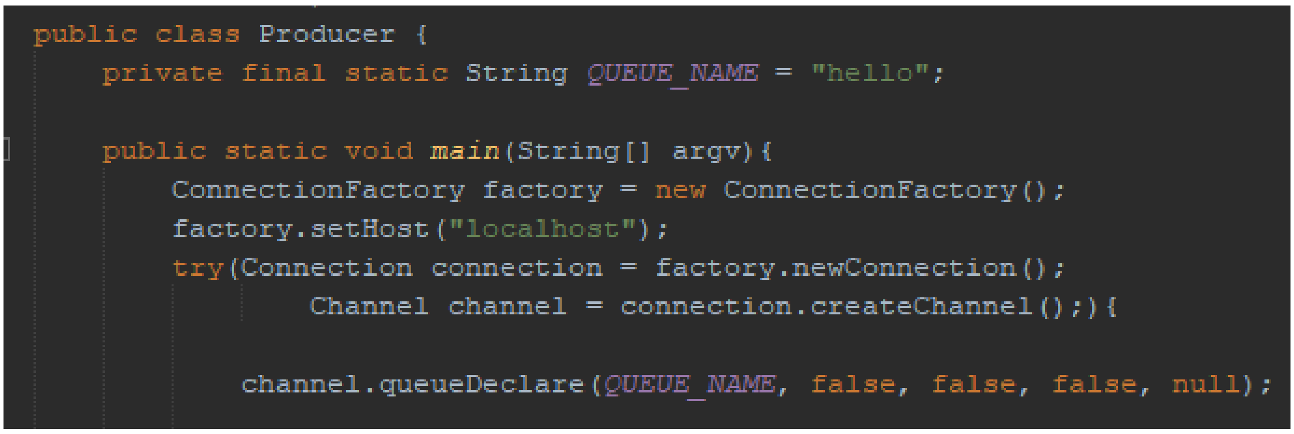


```java
public class myListenerQuotazioni implements MessageListener {
    @Override
    public void onMessage(Message message) {
        String final_name = "";
        String final_text = "";
        float final_value = 0;
        TextMessage msg = null;

        if(message instanceof TextMessage ){
            try {
                msg = (TextMessage) message;
                final_name  = msg.getStringProperty("Nome");
                if(!final_name.equals("Mondadori")) return;
                final_value = msg.getFloatProperty("Valore");
                final_text  = msg.getText();

            } catch (JMSException ex) {
                Logger.getLogger(myListenerQuotazioni.class.getName()).log(Level.SEVERE, null, ex);
            }
        }
        System.out.println("[CLIENT LISTENER]\nNAME: "+final_name + " VALUE: "+final_value+"\n"
                + "TEXT: "+final_text);
    }

}
```

LAB4 – RABBITMq

RabbitMQ is used to implement **QUEUE** and **PRODUCER/SUBSCRIBER**.
They are both very similar, there are just a few small differences in a couple of command that distinguish between them, mostly **declareQueue (queue)** and **exchangeDeclare** (p/s)

1)QUEUE



```java
public class Producer {
    private final static String QUEUE_NAME = "hello";

    public static void main(String[] argv){
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        try(Connection connection = factory.newConnection();
                Channel channel = connection.createChannel();){

            channel.queueDeclare(QUEUE_NAME, false, false, false, null);
```

This first part is the same for both, producer and receiver

1.1)Producer

This will create a **QUEUE** named "hello" at 127.0.0.1 with the command

channel.queueDeclare(QUEUE_NAME, false, false, false, null);

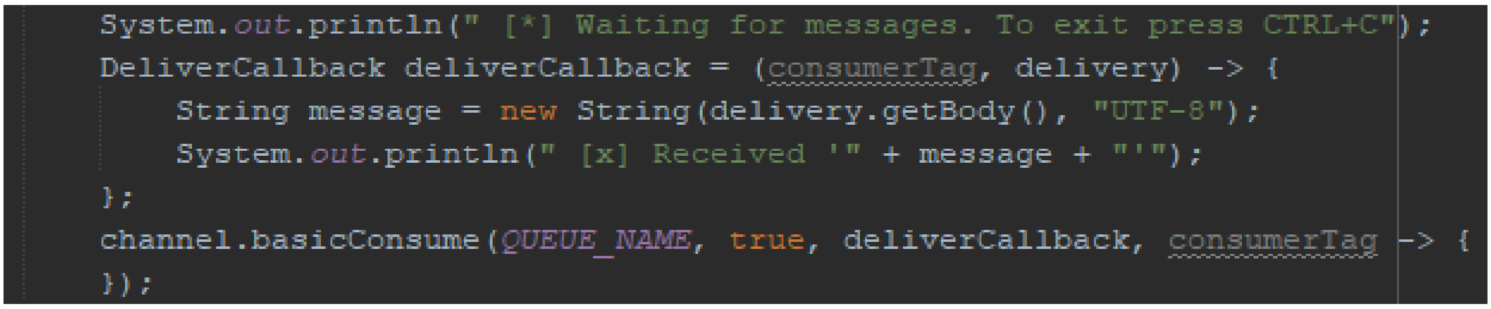
At this point we are able to generate and produce message (ie: inside a while(true) loop) and we can send them with:

channel.basicPublish("", QUEUE_NAME, null, message.getBytes());

//

1.2)Receiver

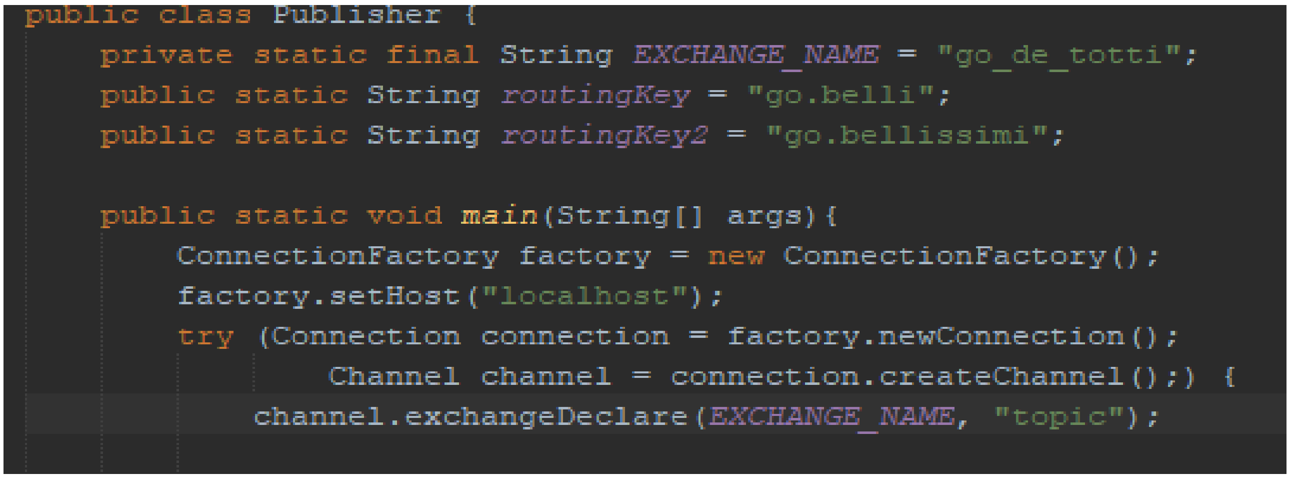
The first part is identical, we have to put the same QUEUE_NAME, the only difference is that for some reason we do not need to put Connection and Channel inside a Try



```java
System.out.println(" [*] Waiting for messages. To exit press CTRL+C");
DeliverCallback deliverCallback = (consumerTag, delivery) -> {
    String message = new String(delivery.getBody(), "UTF-8");
    System.out.println(" [x] Received '" + message + "'");
};
channel.basicConsume(QUEUE_NAME, true, deliverCallback, consumerTag -> {
});
```

Inside the deliveCallback we can add everything we want to be done when a message arrives, in this case we are only printing it but we can do anything.

2)Publisher and Subscriber



```
public class Publisher {
    private static final String EXCHANGE_NAME = "go_de_totti";
    public static String routingKey = "go.belli";
    public static String routingKey2 = "go.bellissimi";

    public static void main(String[] args){
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        try (Connection connection = factory.newConnection();
                Channel channel = connection.createChannel();) {
            channel.exchangeDeclare(EXCHANGE_NAME, "topic");
```

Very similar to the queue, the only difference is that this time we are creating an Exchange. To do it we need an EXCHANGE_NAME and all the ROUTINGKEY we want, they will be the different TOPIC people can subscribe into. (A producer can produce for many topic).

2.1)Publisher

After the inizialization, almost identical to the one of the QUEUE rather than the

channel.exchangeDeclare(EXCHANGE_NAME, "topic");

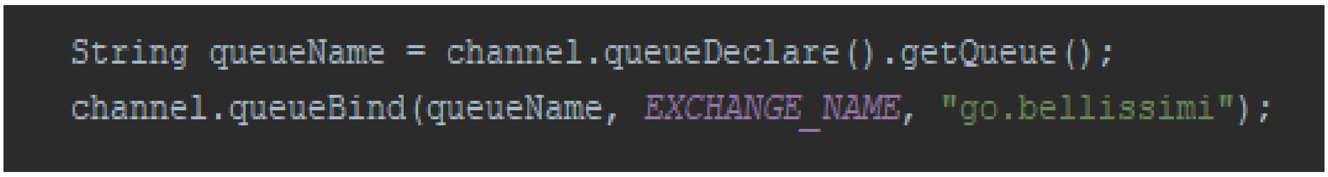
We are ready to publish and send our message to the different topics by using

channel.basicPublish(EXCHANGE_NAME, queue, null, message.getBytes("UTF-8"));

^Topic name = routingKey

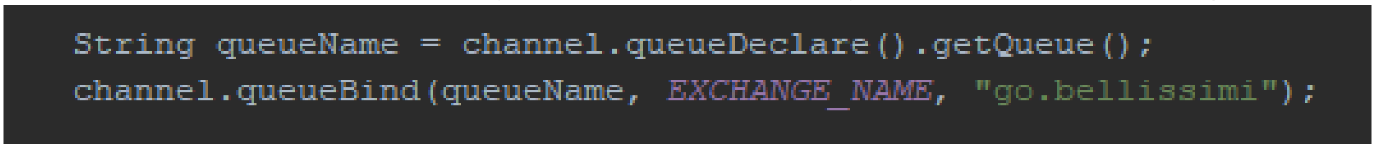
2.1)Subscriber-i

Again, identical to the one above, however this time we cant directly ask for the messages but we have to specify the topic we are interested into.



```
String queueName = channel.queueDeclare().getQueue();
channel.queueBind(queueName, EXCHANGE_NAME, "go.bellissimi");
```

After that we use the same sequence to receive and manipulate each received messages.



```
String queueName = channel.queueDeclare().getQueue();
channel.queueBind(queueName, EXCHANGE_NAME, "go.bellissimi");
```

[EXTRA?] Databases

You usually rely on a specific class with a specific main that is used to clean and create a .db file that will then be used both in read but also in write by other classes, object.

Preparation is very Simple:

```java
public class Database {

    public static void main(String[] args) throws ClassNotFoundException, SQLException {
        Class.forName("org.sqlite.JDBC");

        Connection connection = null;
        try {
            connection = DriverManager.getConnection(
                "jdbc:sqlite:C:/Users/Giulio Garzia/Documents/NetBeansProjects/Esame2019.07/database.db");

            Statement statement = connection.createStatement();
            statement.setQueryTimeout(30);
```

We create a **connection** to the databases and on that connection we create a **statement** with a timeout of 30.

Now, let's assume that in this example we have 2 tables:

```
DIRECTORS: ID  <-------------------------------->  MOVIES:  ID
           name                                              directorID
           yearOfBirth                                       title
                                                             year
```

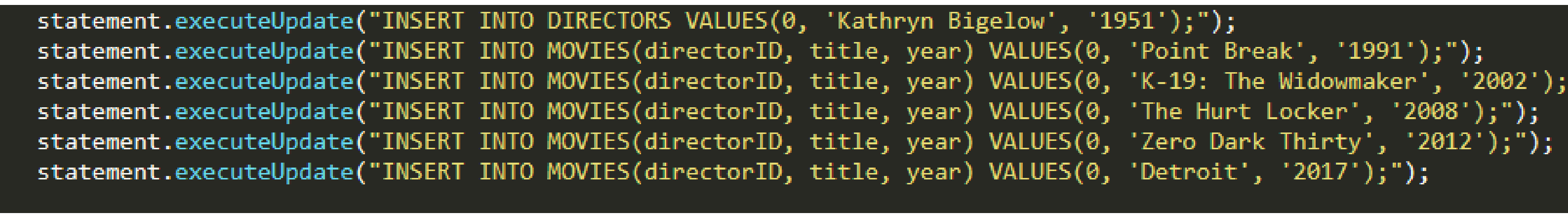
The first thing you want to do is to clear any possible precedent creation of those table

```java
statement.executeUpdate("DROP TABLE IF EXISTS DIRECTORS;");
statement.executeUpdate("DROP TABLE IF EXISTS MOVIES;");
```

And then you want to create the two tables discussed before;
also setting directorID as foreign key from MOVIES -> DIRECTORS (I don't remember shit about db).

```java
statement.executeUpdate("CREATE TABLE DIRECTORS (ID INTEGER PRIMARY KEY AUTOINCREMENT, "
        + "name STRING, "
        + "yearOfBirth STRING);");
statement.executeUpdate("CREATE TABLE MOVIES(ID INTEGER PRIMARY KEY AUTOINCREMENT, "
        + "directorID STRING, "
        + "title STRING, "
        + "year STRING, "
        + "FOREIGN KEY(directorID) REFERENCES DIRECTORS(ID));");
```

Finally we can start populate the database, adding all the wanted entries:



```
statement.executeUpdate("INSERT INTO DIRECTORS VALUES(0, 'Kathryn Bigelow', '1951');");
statement.executeUpdate("INSERT INTO MOVIES(directorID, title, year) VALUES(0, 'Point Break', '1991');");
statement.executeUpdate("INSERT INTO MOVIES(directorID, title, year) VALUES(0, 'K-19: The Widowmaker', '2002');
statement.executeUpdate("INSERT INTO MOVIES(directorID, title, year) VALUES(0, 'The Hurt Locker', '2008');");
statement.executeUpdate("INSERT INTO MOVIES(directorID, title, year) VALUES(0, 'Zero Dark Thirty', '2012');");
statement.executeUpdate("INSERT INTO MOVIES(directorID, title, year) VALUES(0, 'Detroit', '2017');");
```

Here we are adding a DIRECTOR, giving all the 3 fields (id, name, year), so there is no need to specify anything.
While adding the MOVIE we left the ID value (maybe because it's autoincrement?) we specify what fields we are providing, in this case (directorID, title, year);

Note, a lot of the time when creating a statement you don't know at priori what you are going to write in it, maybe you need to write the input you receive from a queue or a topic, in this kind of situation you do not use the patter we used before.

aka: connection = DriverManager.getConnection("jdbc:sqlite:C:/Users/…);
Statement statement = connection.**createStatement**();
statement.setQueryTimeout(30);
statement.executeUpdate("HERE_THE_ACTUAL_STATEMENT")

But instead you use:
PreparedStatement statement = connection.**prepareStatement**
("INSERT INTO FLIGHTS(flight, status) VALUES(**?**, **?**);");

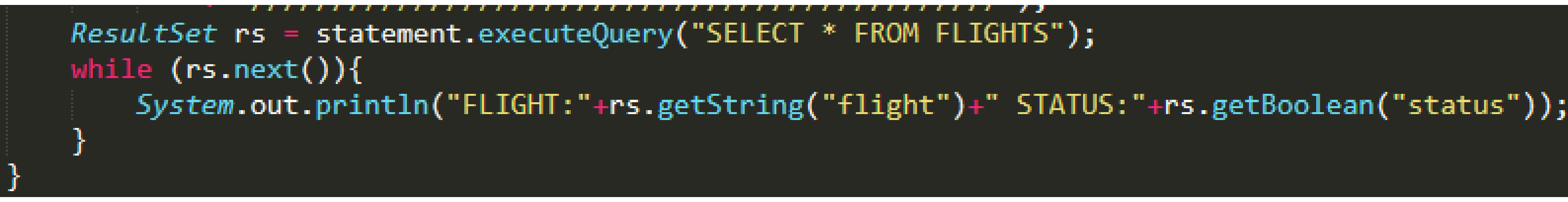
Where the two (**?**) indicate that we are going to manually provide two additional value to that statement by doing:

statement.setString(1, "valuefor1");
statement.setString(2, "valuefor2");
statement.executeUpdate();

But what about when we don't need INSERT/UPDATE/DELETE/DROP/CREATE anything in our database but instead simply execute a query and return a list of results?

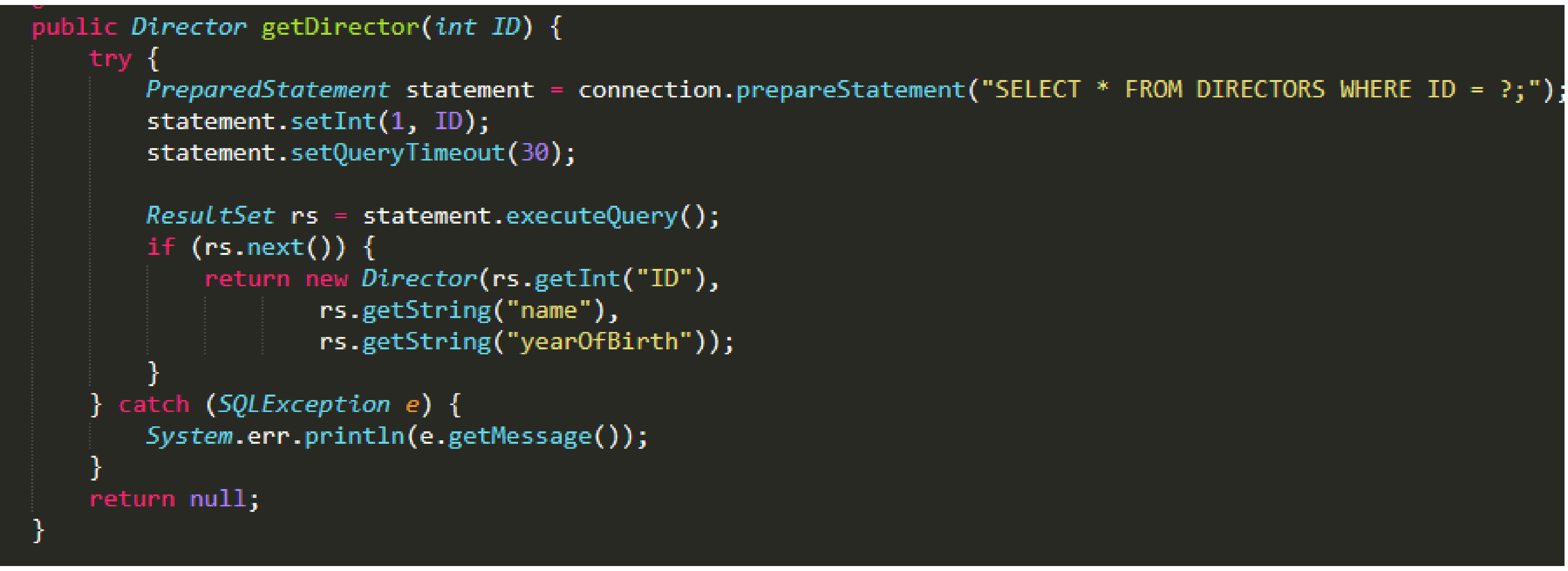
executeUpdate: INSERT/UPDATE/DELETE/DROP/CREATE

executeQuery : SELECT



```java
    ResultSet rs = statement.executeQuery("SELECT * FROM FLIGHTS");
    while (rs.next()){
        System.out.println("FLIGHT:"+rs.getString("flight")+" STATUS:"+rs.getBoolean("status"));
    }
}
```

In general after sending a query you store the result in a ResultSet variable, and you iterate trought it with a while(rs.next), [or a if(rs.next) if you are expecting only one result].
Whay you receive can be seen as a row, containing all the attributes specified in the SELECT *.
In this case we are interested in a String attribute named flight and a Boolean named status.



```java
public Director getDirector(int ID) {
    try {
        PreparedStatement statement = connection.prepareStatement("SELECT * FROM DIRECTORS WHERE ID = ?;");
        statement.setInt(1, ID);
        statement.setQueryTimeout(30);

        ResultSet rs = statement.executeQuery();
        if (rs.next()) {
            return new Director(rs.getInt("ID"),
                    rs.getString("name"),
                    rs.getString("yearOfBirth"));
        }
    } catch (SQLException e) {
        System.err.println(e.getMessage());
    }
    return null;
}
```

NEEDED POM.XML

1)SOAP:

```
 <dependency>
          <groupId>org.apache.cxf</groupId>
          <artifactId>cxf-rt-frontend-jaxws</artifactId>
          <version>3.1.6</version>
</dependency>
<dependency>
          <groupId>org.apache.cxf</groupId>
          <artifactId>cxf-rt-transports-http-jetty</artifactId>
          <version>3.1.6</version>
</dependency>
```

2)REST:

```
<dependency>
          <groupId>org.apache.cxf</groupId>
          <artifactId>cxf-rt-frontend-jaxrs</artifactId>
          <version>3.1.7</version>
</dependency>
<dependency>
          <groupId>org.apache.cxf</groupId>
          <artifactId>cxf-rt-transports-http-jetty</artifactId>
          <version>3.1.7</version>
</dependency>
<dependency>
          <groupId>org.apache.httpcomponents</groupId>
          <artifactId>httpclient</artifactId>
          <version>4.5.2</version>
</dependency>
```

3)JMS

```
<dependency>
        <groupId>org.apache.activemq</groupId>
        <artifactId>activemq-all</artifactId>
        <version>5.14.5</version>
</dependency>
```

4)Rabbit

```
<dependency>
         <groupId>com.rabbitmq</groupId>
        <artifactId>amqp-client</artifactId>
        <version>5.7.2</version>
</dependency>
```

5)Database

```
<dependency>
        <groupId>org.xerial</groupId>
        <artifactId>sqlite-jdbc</artifactId>
        <version>3.7.2</version>
</dependency>
```

EXTRA) Some java releted String manipulations stuff.

Array vs ArrayList (eg of String)->

String[] s = new String[NumOfElementsOfTheArray] //**Array**:you need to know the num of elements

List<String> s = new **ArrayList**<>(); // ArrayList, you can add item one by one, no fixed size.

Note) On both of them you can use **.size()** to get the num of elements.

ONLY on the ArrayList you can use **.containts(elem) => boolean**

substring(int i): return a string that is a substring, starting from index i to the end.

substring(int i1, int i2): as above, but the stirng is the one between index i1 and i2

indexOf(String s): Will return the index of the first occurrence of the string s.
^A possible use could be: we have a string s = "Giulio,Garzia" we want to separate them.

```
 String name = s.substring(0, indexOf(',')) 
 String surname = s.substring( indexOf(',')+1)
```

split(String s): Return an array of strings, splitted according to the value of s.
->”Ciao come stai”.split(“ ”) ==> [“Ciao”,”come”,”stai”]

For each operator:
Once you have an ArrayList, example -> List<String> clients_list = new ArrayList<>();
You can scan it by writing:

```
                    for(String client : clients list){
                            //During each iteration of the for, client will take the value
                            //of a value inside the ArrayList
}
```