Cours - Intergiciels - S7

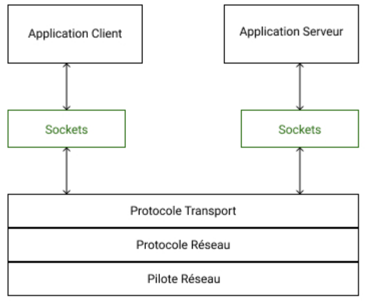
> Écrit par Clément Lizé

1 - Sockets

1.1 - Contexte

Sockets = API (Interface de programmation)

Point de communication → couche applicative



1.2 - Deux modes

Mode connecté : TCP

➡️ Le plus utilisé

✔️ Connexion interrompue → applications informées

👌 Très fiable

😞 Plus lent

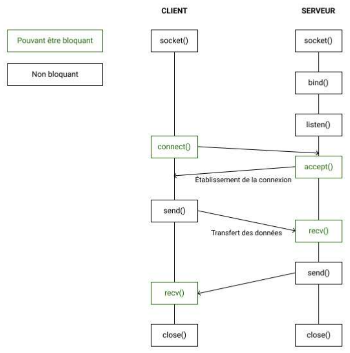
Mode non connecté : UDP

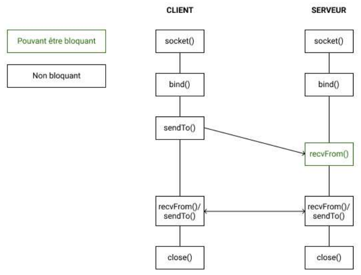
➡️ Moins utilisé que TCP

✔️ Données envoyées en paquets indépendants

👌 Plus rapide car pas de connexion

😞 Moins fiable



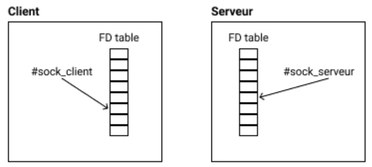


1.3 - Fonctions en C

1.3.1 - socket()

➡️ Créer un socket

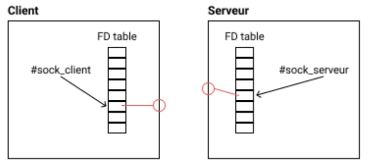
📄 `int socket (int family, int type, int protocol)`



1.3.2 - bind()

➡️ Créer un lien entre un socket et un port machine

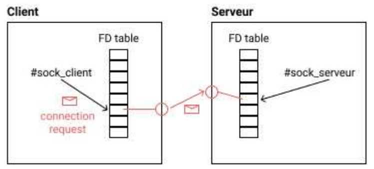
📄 `int bind (int sock_desc, struct sockaddr *my_@, int lg_@)`



1.3.3 - connect()

➡️ Envoyer une demande de connexion au serveur (depuis le client)

📄 `int connect (int sock_desc, struct sockaddr *my_@, int lg_@)`



1.3.4 - listen()

➡️ Utiliser le socket (côté serveur) pour recevoir des demandes de connexion

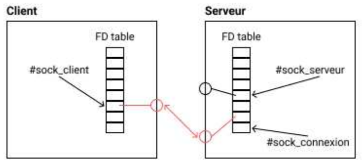
📄 `int connect (int sock_desc, int nbr)`

🚨 `nbr` = nombre de connexions qui peuvent être en attente

1.3.5 - accept()

➡️ Côté serveur. Bloque les demandes de connexion, crée un nouveau socket de communication puis ré-autorise les demandes de connexions. Le nouveau socket sert pour la communication entre le client et le serveur et sera détruit à la fin de celle-ci.

📄 `int accept (int sock_desc, struct sockaddr *client, int lg_@)`



1.4 - Les sockets en Java

1.4.1 - DNS

```
import java.net.*;

public class Enseeiht1 {

  public static void main (String[] args) {

    try {
      InetAddress address = InetAddress.getByName("www.enseeiht.fr");
      System.out.println(address);
      // OU
      InetAddress a = InetAddress.getLocalHost();
      System.out.println(a.getHostName() + " / " + a.getHostAddress());
    }
    catch (UnknownHostException e) {
      System.out.println("cannot find www.enseeiht.fr");
    }
  }
}
```

➡️ InetAddress = classe de java.net

➡️ getByName : transforme un nom de machine en adresse IP

➡️ getLocalHost : récupère l'adresse Inet de localhost

1.4.2 - Mode connecté TCP

a - Code client

```
class Client {
  try {
    Socket s = new Socket("www.enseeiht.fr",80);
    InputStream is = s.getInputStream();
    OutputStream os = s.getOutputStream();
  }
  catch (UnknownHostException u) {
    System.out.println("Unknown host");
  }
  catch (IOException e) {
    System.out.println("IO exception");
  }
}
```

➡️ Socket : objet Socket, initialisé avec une adresse (nom ou IP) et un port. Équivalent en C de *socket()* et *connect()*

➡️ InputStream : permet de lire des octets

➡️ OutputStream : permet d'écrire des octets

b - Code serveur

```
class Serveur {
  try {
    ServerSocket ss = new ServerSocket(80);
```

➡️ ServerSocket : Socket qui a pour objectif de recevoir des demandes de connexion. Équivalent en C de *socket()*, *bind()* et *listen()*

```
    Socket s = ss.accept();
    InputStream is = s.getInputStream();
    OutputStream os = s.getOutputStream();
  }
  catch (UnknownHostException u) {
    System.out.println("Unknown host");
  }
  catch (IOException e) {
    System.out.println("IO exception");
  }
}
```

➡️ accept : Retourne un socket de communication avec le client. Ce nouveau socket est indépendant de *ss*, ainsi le serveur peut à nouveau recevoir des demandes de connexion pendant qu'il échange avec le client sur le socket s.

➡️ InputStream : permet de lire des octets

➡️ OutputStream : permet d'écrire des octets

c - Fonctions utiles

➡️ `InputStreamReader` ou `OutputStreamReader` : convertit un stream d'octets en un stream de caractères

➡️ `BufferedReader` : implémente la notion de tampon : permet la lecture caractère par caractère, ligne par ligne, ..

➡️ `PrintWriter` : permet d'afficher (avec println par ex)

1.4.3 - Mode non connecté UDP

a - Code client

```
class Client {
  try {
    int p = 8888; // for receiving a response
    byte[] t = new byte[10];

    FileInputStream f = new FileInputStream("data.txt");
    int r = f.read(t);

    DatagramSocket s = new DatagramSocket(p);
    DatagramPacket d = new DatagramPacket(t, r,
      InetAddress.getByName("thor.enseeiht.fr"), 9999);
    s.send(d);
  }
  catch (Exception e) {
    System.err.println(e);
  }
}
```

➡️ DatagramSocket : permet de créer un socket associé à un port. Équivalent en C de *socket()* et *bind()*

➡️ DatagramPacket : un packet prêt à l'envoi, créé à partir d'un tableau d'octets. *r* est la taille des données qu'on lit du fichier.

➡️ send : envoi du packet

b - Code Serveur

```
class Serveur {
  try {
   int p = 9999;
   byte[] t = new byte[10];

   DatagramSocket s = new DatagramSocket(p);
   DatagramPacket d = new DatagramPacket(t,t.length);

   s.receive(d);

   String str = new String(d.getData(), 0, d.getLength());
   System.out.println(d.getAddress()+"/"+d.getPort()+"/"+str);

  }
  catch (Exception e) {
   e.printStackTrace();
  }
}
```

➡️ DatagramSocket : permet de créer un socket associé à un port. Équivalent en C de *socket()* et bind()

➡️ DatagramPacket : un packet prêt à l'envoi, créé à partir d'un tableau d'octets. *r* est la taille des données qu'on lit du fichier.

➡️ receive : lit sur le socket UDP et stocke les données dans le DatagramPacket

➡️ getData : renvoie un tableau d'octets depuis le DatagramPacket

➡️ getLength : retourne la taille des données reçues

➡️ getAddress et getPort : renvoient les adresse et port du client pour pouvoir envoyer une réponse

1.4.4 - Exemple complet

```
public class Client {
  public static void main (String[] str) {
    try {
      Socket csock = new Socket("localhost",9999);
      ObjectOutputStream oos = new ObjectOutputStream (
        csock.getOutputStream());
      oos.writeObject(new Person("Dan","Hagi",53));
      csock.close();
    }
    catch (Exception e) {
      System.out.println("An error has occurred ...");
    }
}
```

```
public class Server {
  public static void main (String[] str) {
    try {
      ServerSocket ss;
      int port = 9999;
      ss = new ServerSocket(port);
      System.out.println("Server ready ...");
      while (true) {
        Slave sl = new Slave(ss.accept());
        sl.start();
      }
    }
    catch (Exception e) {
      System.out.println("An error has occurred ...");
    }
  }
}
```

```
public class Slave extends Thread {
  Socket ssock;

  public Slave(Socket s) {
    this.ssock = s;
  }

  public void run() {
    try {
      ObjectInputStream ois = new ObjectInputStream(
        ssock.getInputStream());
      Person v = (Person)ois.readObject();
      System.out.println("Received person: "+ v.toString());
      ssock.close();
    }
    catch (Exception e) {
      System.out.println("An error has occurred ...");
    }
  }
}
```

2 - RPC et Java-RMI

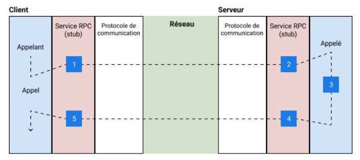
2.1 - Contexte

RPC = "Remote Procedure Call". Objectif = appels locaux et distants avec la même syntaxe

RMI = "Remote Method Invocation" = API qui implémente RPC en Java

2.2 - RPC

2.2.1 - Fonctionnement général



➡️ En bleu : segments de code avec le client qui appelle un service, et le serveur qui fournit le service

➡️ En rouge : segments de code créés par RPC : le stub client et le stub serveur

👉 Le stub client donne l'illusion que le service fourni par le serveur est local. Pour créer cette illusion, le client implémente les même procédures que le serveur

1️⃣ Le stub **collecte** les paramètres, **les assemble** dans un message et **arme un timer**
Un **identifiant** est généré pour l'appel RPC

2️⃣ Le protocole de transport **délivre** le message vers le service RPC (le stub serveur)
Le stub serveur **désassemble** les paramètres, l'id RPC est enregistré

3️⃣ **Le code est exécuté**

4️⃣ Les paramètres du résultat sont **assemblés** dans un message
Le stub serveur **arme un timer** et **transmet** le message au protocole de communication

5️⃣ Le protocole de transport **délivre** le message vers le service RPC (le stub client)
Le stub client **désassemble** les paramètres du résultat
Un **ACK** est envoyé au serveur, qui **désarme son timer**
Les **résultats sont transmis** à l'appelant

2.2.2 - Les rôles des stubs

Stub client

Sert d'interface avec le client

- Reçoit les appels locaux
- Les transforme en un appel distant
- Reçoit les résultats
- Retourne les résultats avec une procédure de retour normale

Stub serveur

Exécute sur le serveur

- Reçoit les appels sous forme de messages
- Exécute la procédure
- Reçoit les résultats de la procédure en local
- Transmets les résultats sous forme de message

2.3 - RMI

Objectif : rendre transparent la manipulation d'objets situés dans une autre JVM (souvent sur une autre machine)

2.3.1 - Service de nommage : RMI registry

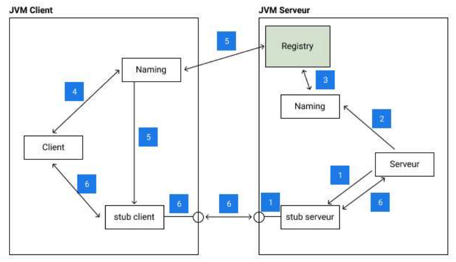
Registry = annuaire des services qui tournent sur le serveur distant

🚨 Contrainte : doit se situer sur la même machine que le serveur

➡️ Pour ajouter un service : *bind()*

➡️ Pour consulter un service : *lookup()*

2.3.2 - Fonctionnement général



1️⃣ Le serveur crée le stub serveur et le port est ouvert

2️⃣ Le serveur s'enregistre auprès du Naming (méthode *bind()* ou *rebind()*)

3️⃣ Le Naming enregistre dans le registry le nom de l'objet serveur, son stub et les coordonnées de connexion (IP et port)

4️⃣ Le client fait appel à son Naming pour localiser le serveur (méthode *lookup()*)

5️⃣ Le naming récupère les coordonnées du serveur et crée le stub client

6️⃣ Le client appelle les méthodes du serveur. Ces requêtes passent par le stub client qui crée un port de communication côté client

2.3.3 - En pratique

```
import java.rmi.*;

public interface monServeur extends Remote {

  public int maMethode() throws RemoteException;

}
```

➡️ Un serveur doit avoir une interface qui hérite de la classe *Remote*.

➡️ Seules les méthodes décrites dans l'interface seront accessibles à distance.

➡️ Toutes les méthodes doivent propager une *RemoteException*.

➡️ Les types de retour peuvent être seulement des :

- Types primitifs (int, boolean, ..)
- Objets distants
- Objets sérialisables (voir plus bas)

🚨 En RMI, tous les arguments sont transmis par copie → un objet n'est pas modifié localement si il a été passé en paramètre d'une méthode distante

```
import java.rmi.server.*;
import java.rmi.*;

public class monServeurImpl extends UnicastRemoteObject implements monServeur {

  public monServeurImpl() throws RemoteException {
    System.out.println("Serveur créé");
  }

  public int maMethode() throws RemoteException {
    return 0;
  }
}
```

➡️ Les serveurs RMI doivent hériter de la classe *UnicastRemoteObject*.

➡️ Toutes les méthodes doivent propager une *RemoteException*, y compris le constructeur.

```
public void main(String[] args) {
  try {
    Naming.bind("//localhost/monService", new monServeurImpl());
  }
  catch (Exception e) {
    e.printStackTrace();
  }
}
```

Ce code doit être exécuté sur la machine serveur. Il permet de déclarer le serveur auprès du Naming (et donc du registry) pour pouvoir l'utiliser avec le client.

➡️ *bind()* permet de déclarer le serveur au naming

➡️ Par défaut, le port du registry est à 1099

```
import java.rmi.*;

public class Client {

  public static void main(String[] args) {

    try {

      monServeur s = (monServeur) Naming.lookup("//localhost/monService");
      System.out.println(s.maMethode());  // 0
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

➡️ La méthode *lookup()* renvoie un stub qui doit être converti en l'interface du serveur. ⚠️ Ne doit pas être convertie en serveur.

➡️ l'appel des méthodes distantes est maintenant aussi simple qu'en local

2.3.4 - Sérialisation

Les objets qui transitent doivent être sérialisables. Quasiment toutes les classes de base le sont, donc pas besoin de s'occuper de ça quand on renvoie un int par exemple. Si on renvoie un objet qu'on a créé, il faut le rendre sérialisable :

```
import java.io.*;

public interface SFiche extends Serializable {

  public String getNom();
}
```

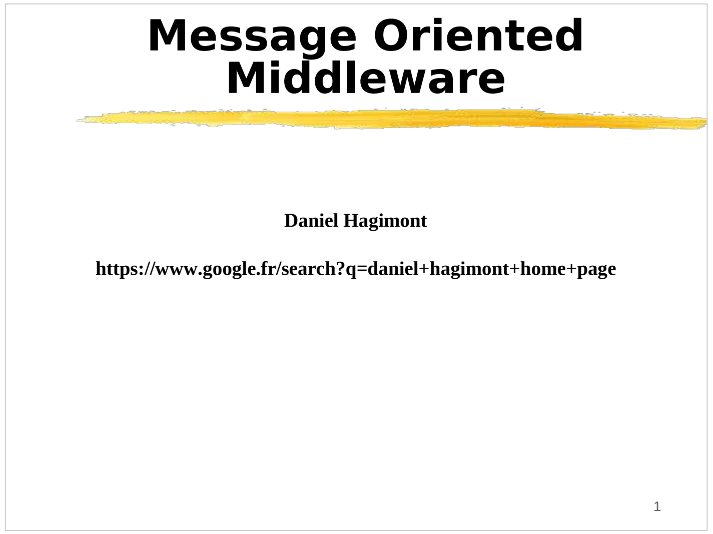
```
import java.rmi.*;

public class SFicheImpl implements SFiche {

  private String nom;

  public SFicheImpl (String nom) {
    this.nom = nom;
  }

  @Override
  public String getNom() {
    return this.nom;
  }
}
```

3 - Web Services

Flemme de résumer, voir cours

4 - JRM

Flemme de résumer, voir cours



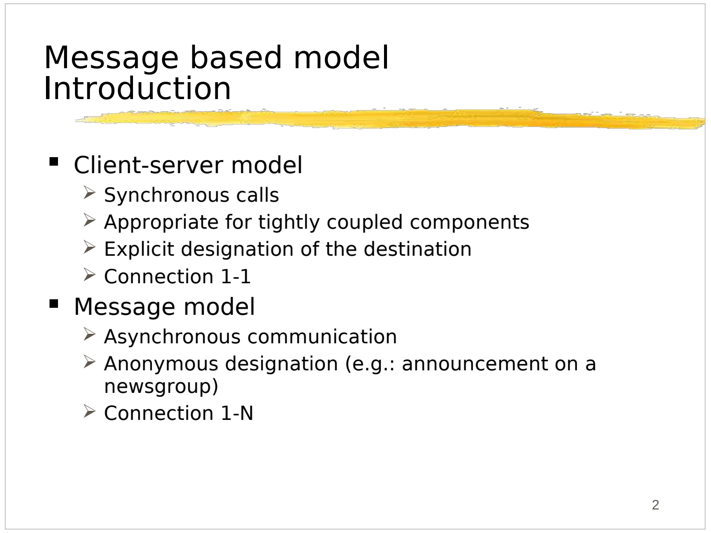
Message Oriented Middleware

Daniel Hagimont

https://www.google.fr/search?q=daniel+hagimont+home+page

1

This lecture is about messaging services that are provided in the context of Message Oriented Middleware (MOM).



Message based model Introduction

- Client-server model
  - Synchronous calls
  - Appropriate for tightly coupled components
  - Explicit designation of the destination
  - Connection 1-1
- Message model
  - Asynchronous communication
  - Anonymous designation (e.g.: announcement on a newsgroup)
  - Connection 1-N

2

For the moment, we can consider that the message model consists in programming distributed applications with simple message exchanges.

The message model has fundamentally different properties compared to the client-server model.

The client-server model :

- relies on synchronous calls (with a request and a response, the client being suspended waiting for the response)

- is well suited for tightly coupled components, i.e. the caller depends on the service provided by the callee

- there's an explicit designation of the callee by the caller

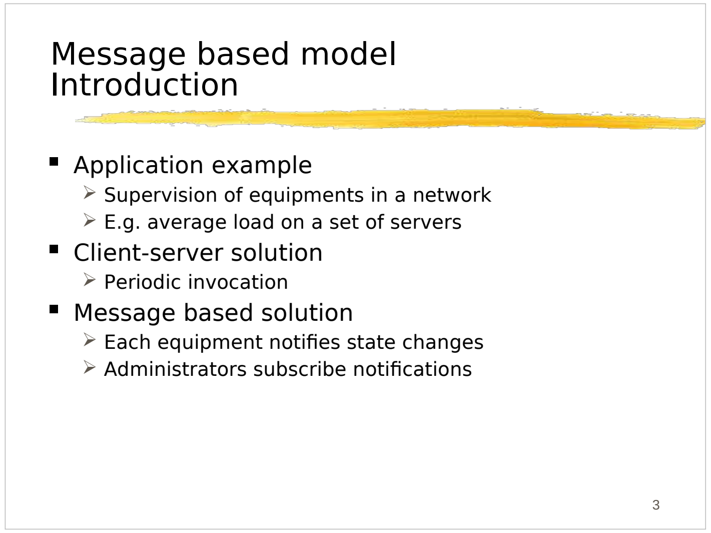
- it's a one to one connection

In opposition, the message model :

- relies on asynchronous communications (the sender does not wait for a response)

- there can be a anonymous designation (when you send a message to anybody who may be interested like an announcement on a newsgroup)

- it's can be a one to many connection



Message based model Introduction

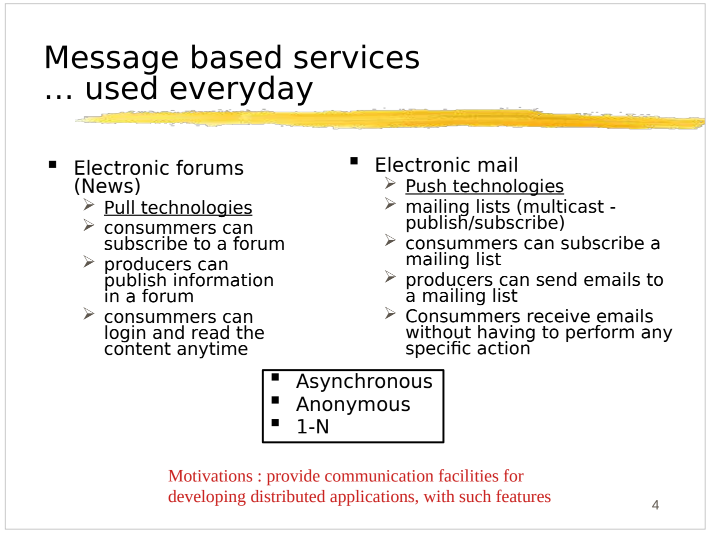
- Application example
  - Supervision of equipments in a network
  - E.g. average load on a set of servers
- Client-server solution
  - Periodic invocation
- Message based solution
  - Each equipment notifies state changes
  - Administrators subscribe notifications

3

We give here an example of application where the message model is better suited. Let's consider the supervision of equipments in a cluster (e.g. the load of the cluster's machines).

A client-server based solution would require a central server performing periodic invocations of all the servers in the cluster.

A message based solution would see each server notify the central server whenever the load changes.



Message based services … used everyday

- Electronic forums (News)
  - Pull technologies
  - consummers can subscribe to a forum
  - producers can publish information in a forum
  - consummers can login and read the content anytime
- Electronic mail
  - Push technologies
  - mailing lists (multicast - publish/subscribe)
  - consummers can subscribe a mailing list
  - producers can send emails to a mailing list
  - Consummers receive emails without having to perform any specific action

- Asynchronous
- Anonymous
- 1-N

Motivations : provide communication facilities for developing distributed applications, with such features

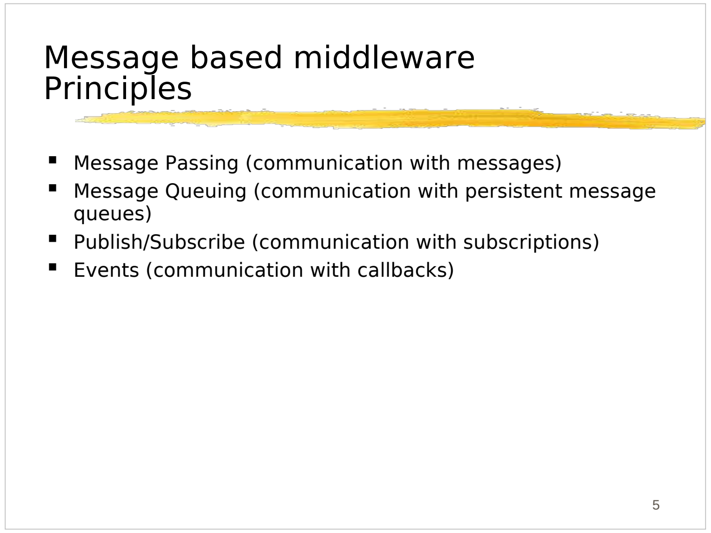
4

The message model is already used for many applications in daily use.

For instance, electronic forums (news) are relying on the message model. Producers publish (send) information on a forum. Consumers subscribe to a forum and read (pull) the information published on the forums they subscribed.

Another example is electronic mail with mailing lists. A producer can send email to a mailing list and consumers can subscribe mailing lists and the messages sent (push) to these mailing lists are received by those consumers.

In both examples, communication is asynchronous, anonymous and may involve several receivers.


Message based middleware Principles

- Message Passing (communication with messages)
- Message Queuing (communication with persistent message queues)
- Publish/Subscribe (communication with subscriptions)
- Events (communication with callbacks)

5

Message based middleware were designed to provide developers with a system support for managing messages and programming distributed applications which exchange messages, with the properties presented previously (asynchronous, anonymous, 1-N).

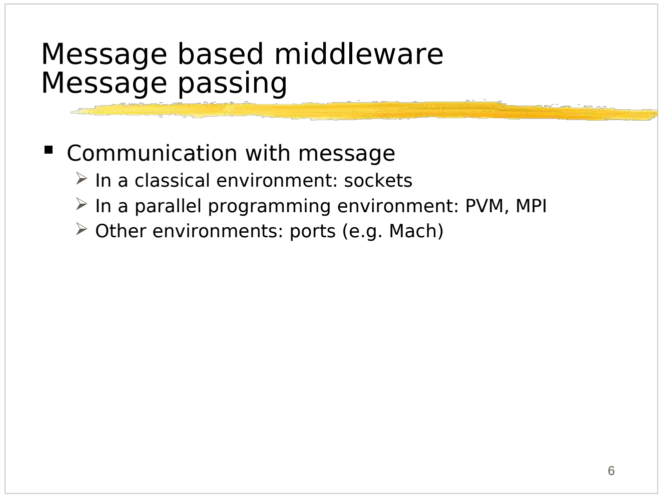
In this context, we distinguish 3 kinds of such messaging service :

- Message passing

- Message queuing

- Publish/subscribe

And one additional service commonly found which is event programming.


Message based middleware Message passing

- Communication with message
  - In a classical environment: sockets
  - In a parallel programming environment: PVM, MPI
  - Other environments: ports (e.g. Mach)

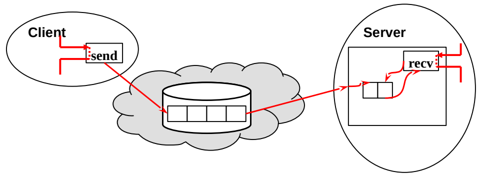
6

Message passing is the simplest service which consists in allowing to send asynchronous messages.

In a classical environment, message passing relies on the socket interface, but it can have other forms, e.g. in an environment devoted to parallel applications (PVM and MPI are parallel environments providing message passing interfaces). Other systems may provides message passing with an interface different from socket (e.g. ports in Mach).

Message based middleware
Message Queuing

- Queue of messages
  - persistent messages (reliability)
- Independence between the emitter and the receiver
  - The receiver is not necessarily active
    => increased asynchronism
  - Several receivers (anonymous)



Message Queuing is the first advanced service which may be provided by a MOM.

The basic difference with message passing is that message queuing provides message persistence.

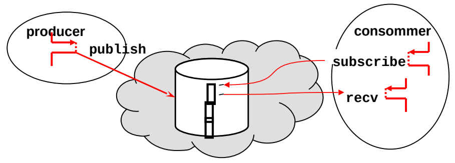
A queue may be allocated and used by clients (producers) or servers (consumers). The queue is managed in the network, meaning that it is not managed in clients neither servers. It is instead managed on machines managed in the middle, i.e. by the message middleware.

Messages are persistent in the sense that we don't require the producer and the consumer to be active at the same time for sending a message (which is the case for message passing). The client may send a message in the queue while the server is inactive (the machine is down). The message will be read by the server at a later time, and may be the client will be inactive at that time.

Another aspect of independence is the fact that a queue may be shared by several producers and consumers. It already provides a sort of anonymous designation.

Message based middleware
Publish/Subscribe

- Anonymous designation
  - The receiver subscribes to a topic
    - Subject-based
    - Content-based
  - The producer sends a message to a topic
- Communication 1-N
  - Several receivers may subscribe



The second advanced service is the publish/subscribe (pub/sub) service.

A receiver may subscribe to a topic.

There are generally 2 types of pub/sub system:

- subject-based : topics are predefined subjects (i.e. subjects have to be created by an administrator)

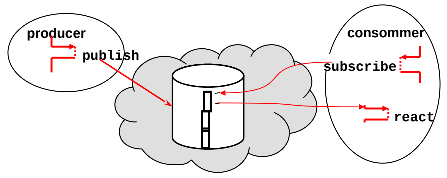
- content-based : topics are filters on the content of messages (e.g. I want to receive messages which include ….)

A producer sends a message to a topic, i.e. either to a given subject or simply with a content. All the receivers who subscribed to the subject, or requested a content which fits with the sent message, will receive a copy of the message.

Here the pub/sub communication service allows message persistence, anonymous designation and multiple receivers.

Message based middleware
Events

- Basic concepts: events, reactions (handling associated with event reception)
- Attachement: association between an event type and a reaction



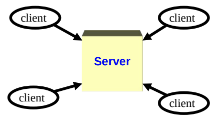
- Exists for all forms of messaging (Message Passing, Message Queuing, Publish/Subscribe)

In order not to have to periodically consult message queues (associated with message queuing or pub/sub) message based middleware often introduces support for event programming.

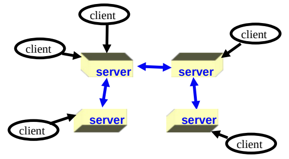
It mainly allows the association between an event (reception of a message) and a reaction (handling program).

Such a facility is available for all forms of communication (message passing, queuing or pub/sub).

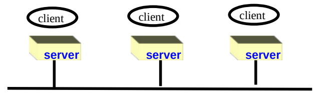
Message based middleware
Implementations



Centralized server
(Hub & spoke)



Distributed servers
(Snow flake)



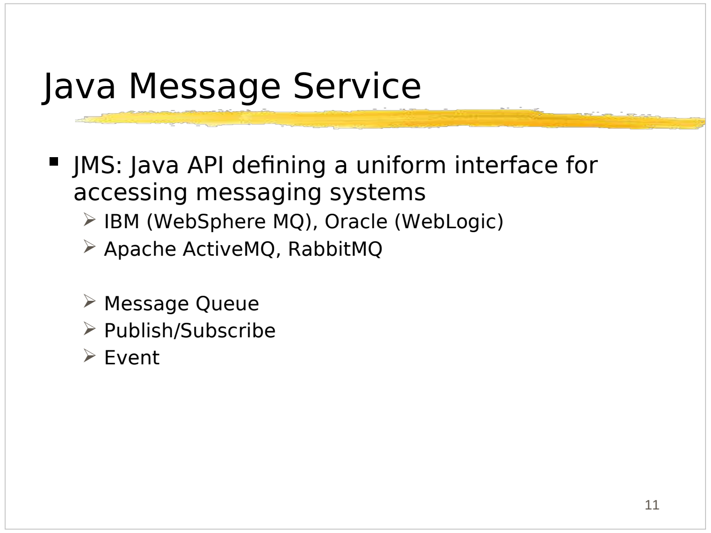
Software bus

Different implementation strategies may be used.

The simplest one is a centralized server remotely used by all clients. This is appropriate for testing, but not for real use as it represents a single-point-of-failure.

Another organization is an interconnection of distributed servers. The interconnection generally depends on the geographic and administrative distribution of clients. The server may implement routing of messages according to the subscriptions from clients.

The last organization is the software bus where all servers know each others. This is generally a strategy used on local (small scale) networks.



Java Message Service

- JMS: Java API defining a uniform interface for accessing messaging systems
  - IBM (WebSphere MQ), Oracle (WebLogic)
  - Apache ActiveMQ, RabbitMQ

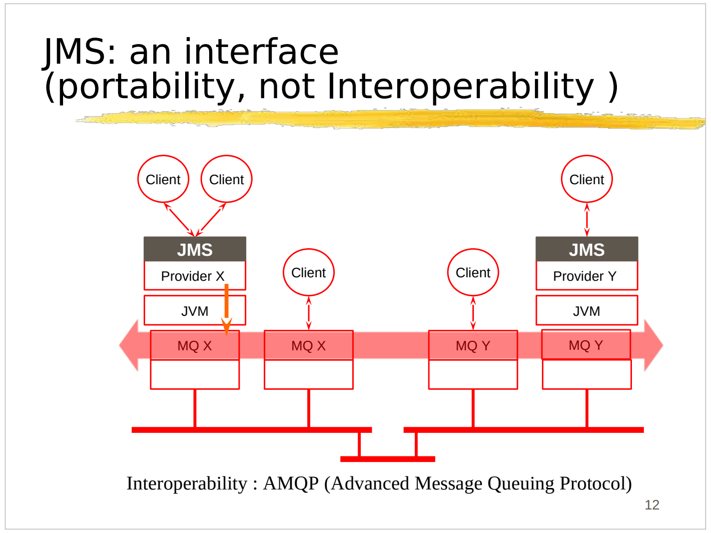
  - Message Queue
  - Publish/Subscribe
  - Event

With the popularity of MOMs, and the development of Java, was proposed a common specification of an API for using a MOM from Java. It should be the same API for all messaging systems (from different providers).

This is JMS for Java Message Service. JMS defines a set of Java interfaces which allows a client to access a messaging system. JMS tries to minimize the concepts to learn and manipulate to use a messaging system, while preserving the diversity of all the existing MOMs.

JMS defines interfaces for managing message Queues and Publish/Subscribe.

JMS: an interface (portability, not Interoperability)

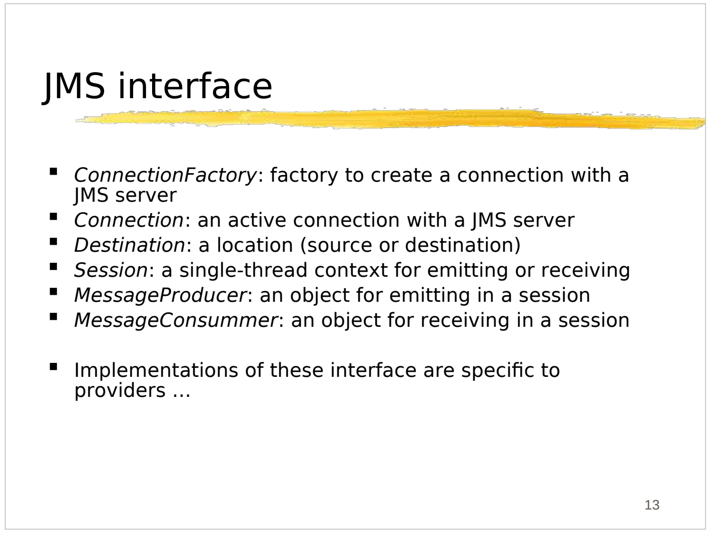


Interoperability : AMQP (Advanced Message Queuing Protocol)

It is important to note that JMS is an interface. Since it is implemented by many MOM providers, it implies that if you implement your applications with JMS, it will run on many MOMs (from different providers). So JMS addressed the issue of the portability of applications.

However, JMS does not bring interoperability. The messages emitted by provider X may have a different format from those emitted by provider Y.

Portability was brought to MOMs with the standardization of AMQP which defines format of exchanged data at the network level.



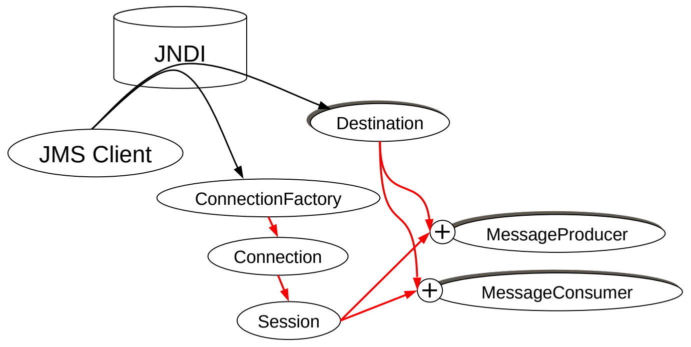
JMS interface

- *ConnectionFactory*: factory to create a connection with a JMS server
- *Connection*: an active connection with a JMS server
- *Destination*: a location (source or destination)
- *Session*: a single-thread context for emitting or receiving
- *MessageProducer*: an object for emitting in a session
- *MessageConsummer*: an object for receiving in a session

- Implementations of these interface are specific to providers ...

JMS may appear complex, but it is rather systematic, and also it had to satisfy all the providers (if the designers wanted all the providers to implement it).

JMS - Architecture



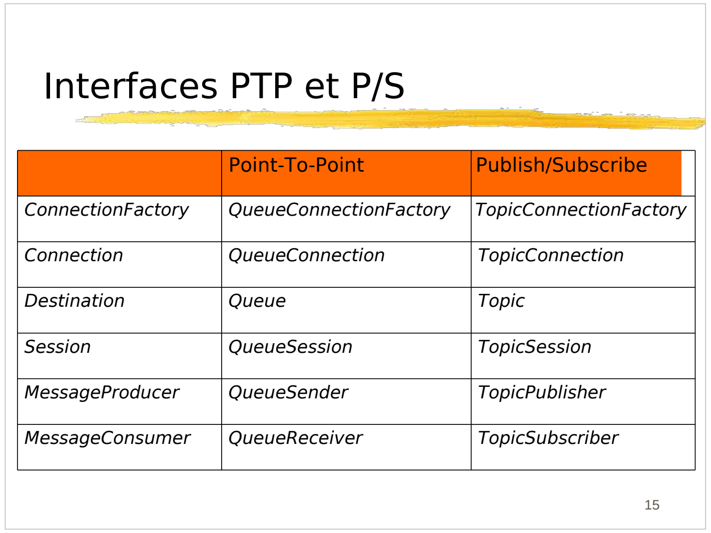
This figure illustrates how these interfaces can be used.

JNDI is the interface of a naming service (such as rmiregistry which is an instance of such a naming service). We assume a JNDI service is available.

A JMS client can obtain from the JNDI service a reference to a ConnectionFactory, which allows to create a Connection (with the JMS server) and then to create a session in this JMS server.

The JMS client can also obtain from the JNDI service a reference to a Destination (an abstract type which can actually refer to a Queue or a Topic).

From a session and a destination, we can create a MessageProducer and a MessageConsumer allowing to emit and receive messages.

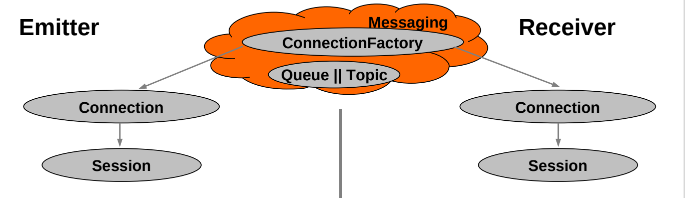


Interfaces PTP et P/S

	Point-To-Point	Publish/Subscribe
ConnectionFactory	*QueueConnectionFactory*	*TopicConnectionFactory*
Connection	*QueueConnection*	*TopicConnection*
Destination	*Queue*	*Topic*
Session	*QueueSession*	*TopicSession*
MessageProducer	*QueueSender*	*TopicPublisher*
MessageConsumer	*QueueReceiver*	*TopicSubscriber*

The interfaces described previously are abstract and are specialized according to the use of message queuing (Point-To-Point) or Publish/Subscribe

JMS - initialization

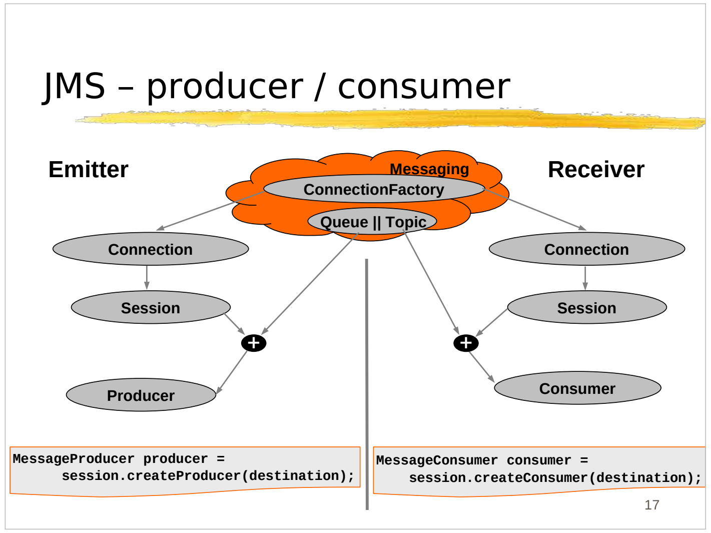


```
ConnectionFactory connectionFactory =
        new ActiveMQConnectionFactory(ActiveMQConnection.DEFAULT_BROKER_URL);
Connection connection = connectionFactory.createConnection();
connection.start();
Session session = connection.createSession(false,Session.AUTO_ACKNOWLEDGE);

Destination destination = session.createQueue("myQueue");
Destination destination = session.createTopic("MyTopic");
```

Here is the code which is common to the emitter and receiver for initializing the connection with the JMS server and obtaining a destination (one of the 2 lines should be chosen, queue or topic ...).

Notice that with ActiveMQ (this is not JMS standard, but specific to ActiveMQ), createQueue() and createTopic() take a URL as parameter, so the same URL used by 2 clients implies the same destination. These ActiveMQ methods correspond to the query of JNDI.

In ActiveMQ, destinations are instantiated at first use.

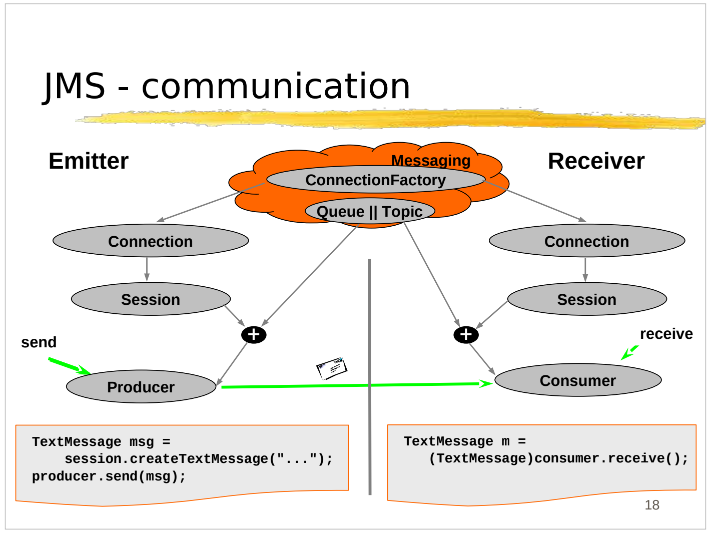


JMS – producer / consumer

```
MessageProducer producer =
      session.createProducer(destination);
```

```
MessageConsumer consumer =
    session.createConsumer(destination);
```

Here, with a session and a destination, we create a producer (left) and a consumer (right).



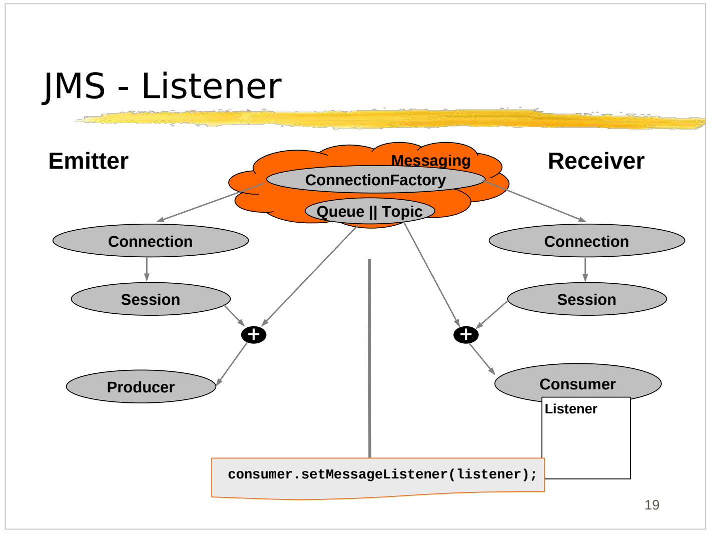
JMS - communication

```
TextMessage msg =
    session.createTextMessage("...");
producer.send(msg);
```

```
TextMessage m =
   (TextMessage)consumer.receive();
```

On the left, we can send a message (here a TextMessage) with a producer.

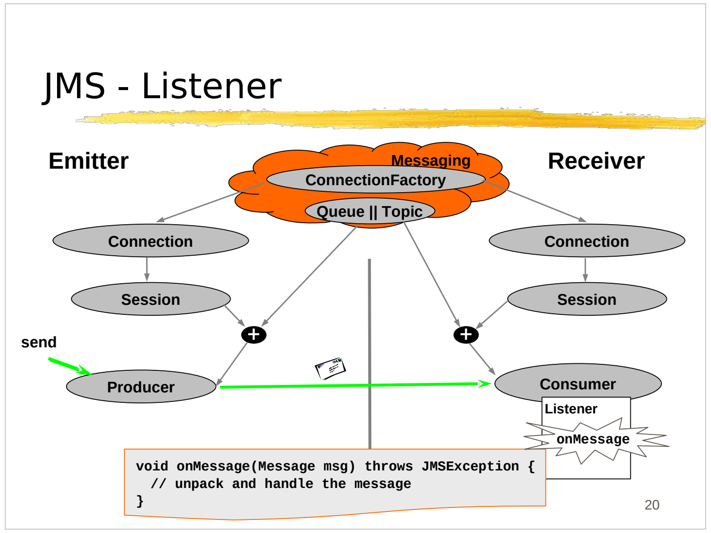
On the right, we can receive a message (here a TextMessage) with a consumer.



JMS - Listener

```
consumer.setMessageListener(listener);
```

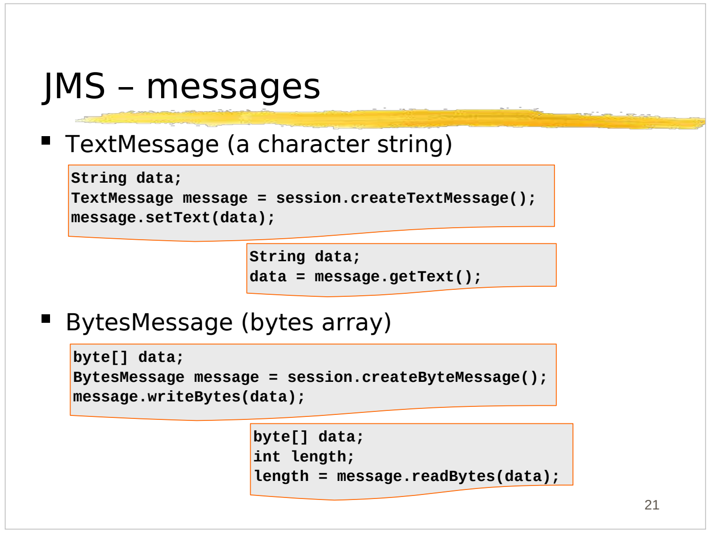
On the consumer side, we can associate a reaction to a message reception event.



JMS - Listener

```
void onMessage(Message msg) throws JMSException {
  // unpack and handle the message
}
```

The registered listener is an instance of a class which implements the onMessage() reaction method.



JMS – messages

- TextMessage (a character string)

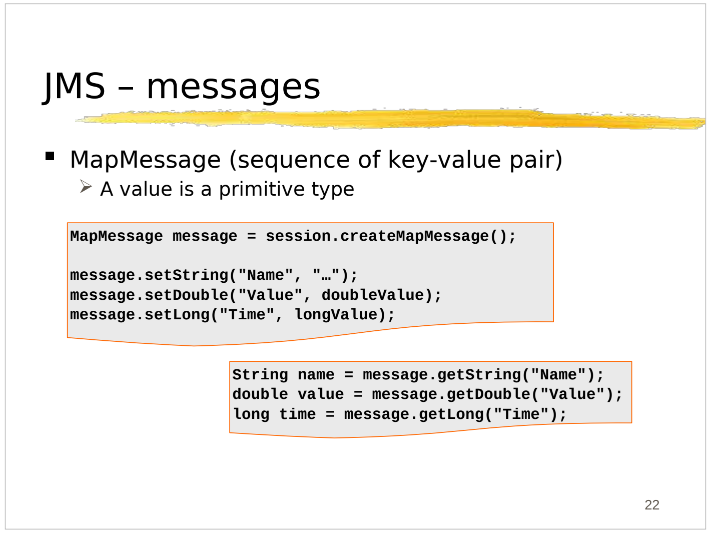
```
String data;
TextMessage message = session.createTextMessage();
message.setText(data);
```

```
String data;
data = message.getText();
```

- BytesMessage (bytes array)

```
byte[] data;
BytesMessage message = session.createByteMessage();
message.writeBytes(data);
```

```
byte[] data;
int length;
length = message.readBytes(data);
```



JMS – messages

- MapMessage (sequence of key-value pair)
  - A value is a primitive type

```
MapMessage message = session.createMapMessage();

message.setString("Name", "…");
message.setDouble("Value", doubleValue);
message.setLong("Time", longValue);
```

```
String name = message.getString("Name");
double value = message.getDouble("Value");
long time = message.getLong("Time");
```

In JMS, messages are types. We can allocate :

- TextMessage (like String)
- BytesMessage (like byte[]).

JMS – messages

- StreamMessage (sequence of values)
  - A value is a primitive type
  - Reading should respect the sequence order to writing

```
StreamMessage message = session.createStreamMessage();

message.writeString("…");
message.writeDouble(doubleValue);
message.writeLong(longValue);
```

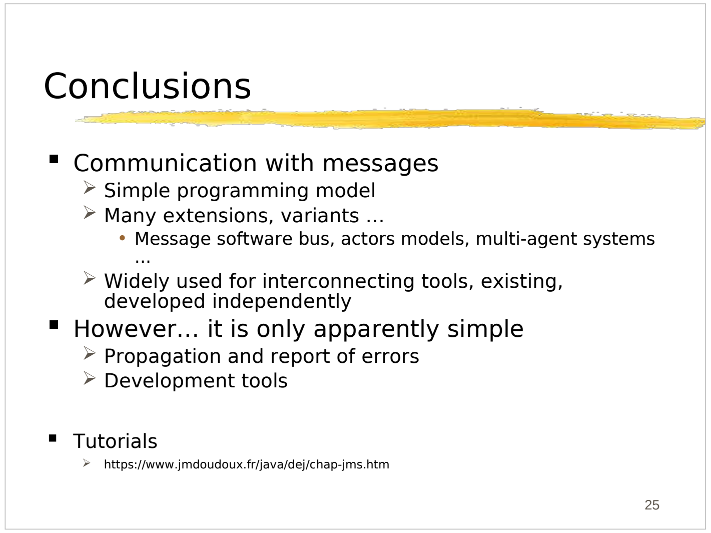
```
String name = message.readString();
double value = message.readDouble();
long time = message.readLong();
```

JMS – messages

- ObjectMessage (serialized objects)

```
ObjectMessage message = session.createObjectMessage();

message.setObject(obj);
```

```
obj = message.getObject();
```



Conclusions

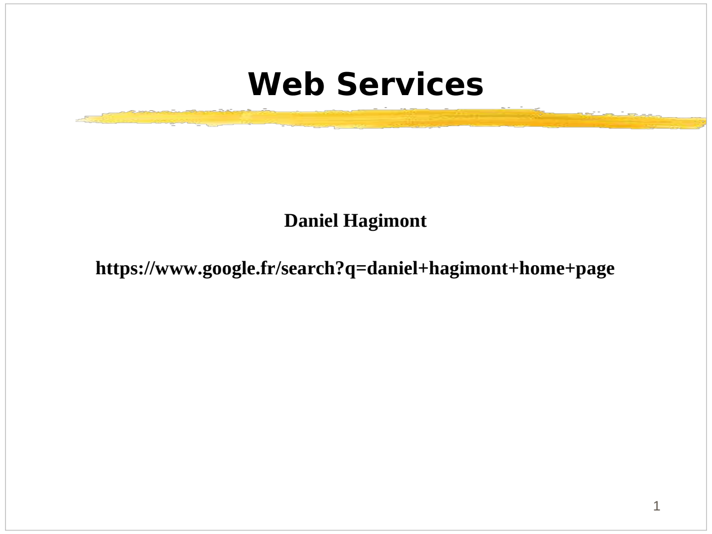
- Communication with messages
  - Simple programming model
  - Many extensions, variants …
    - Message software bus, actors models, multi-agent systems …
  - Widely used for interconnecting tools, existing, developed independently
- However… it is only apparently simple
  - Propagation and report of errors
  - Development tools

- Tutorials
  - https://www.jmdoudoux.fr/java/dej/chap-jms.htm

Even if the message model may seem to be very simple and primitive, many extensions and variants exist.

MOMs are widely used for interconnecting tools, integrating tools that were developed independently.

Notice that simplicity is only apparent, as asynchronism makes it difficult to debug or to have deterministic behaviors.



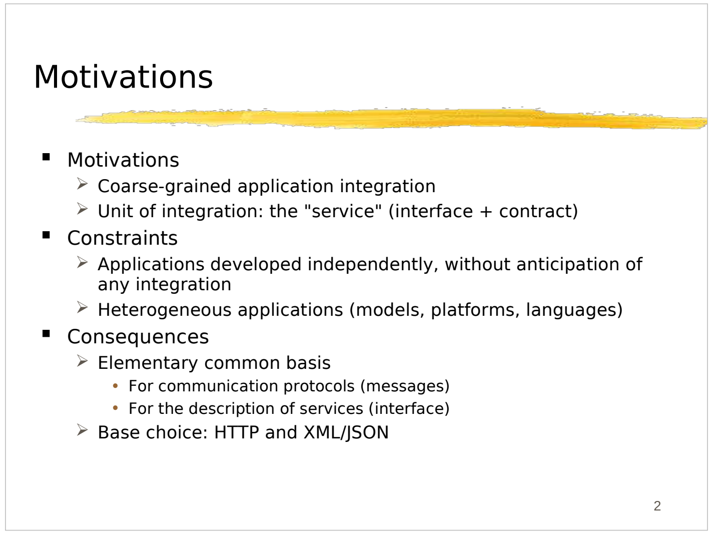
Web Services

Daniel Hagimont

https://www.google.fr/search?q=daniel+hagimont+home+page

1

This lecture is about web services.



Motivations

- Motivations
  - Coarse-grained application integration
  - Unit of integration: the "service" (interface + contract)
- Constraints
  - Applications developed independently, without anticipation of any integration
  - Heterogeneous applications (models, platforms, languages)
- Consequences
  - Elementary common basis
    - For communication protocols (messages)
    - For the description of services (interface)
  - Base choice: HTTP and XML/JSON

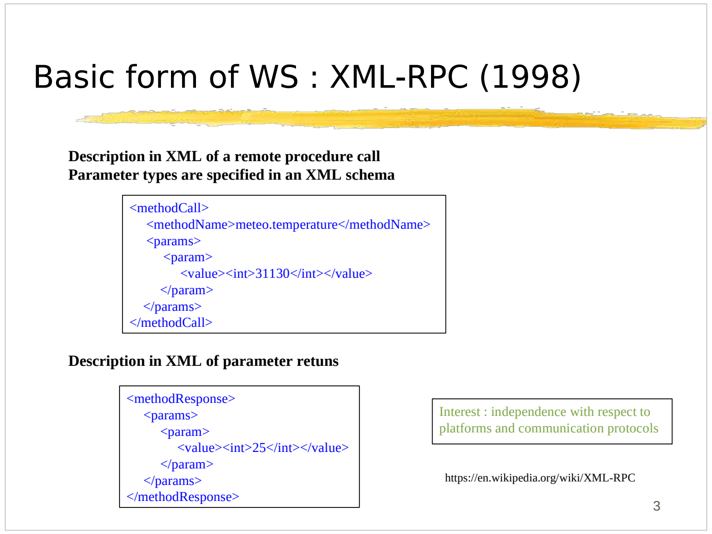
2

The example of RPC tool we have seen, Java RMI, is restricted to interactions within Java applications, allowing remote invocations of Java objects.

With Web services, the motivation is to provide a RPC facility for the interaction (and integration) of coarse-grained applications (that we call services). A service is supposed to be much bigger than a simple Java object.

Web services aim at allowing the interaction between application developed independently, with different environments (models, platforms, languages).

Web services rely on elementary existing protocols and formats: mainly HTTP, XML and JSON.



Basic form of WS : XML-RPC (1998)

Description in XML of a remote procedure call
Parameter types are specified in an XML schema

```
<methodCall>
    <methodName>meteo.temperature</methodName>
    <params>
        <param>
            <value><int>31130</int></value>
        </param>
    </params>
</methodCall>
```

Description in XML of parameter retuns

```
<methodResponse>
    <params>
        <param>
            <value><int>25</int></value>
        </param>
    </params>
</methodResponse>
```

Interest : independence with respect to platforms and communication protocols

https://en.wikipedia.org/wiki/XML-RPC

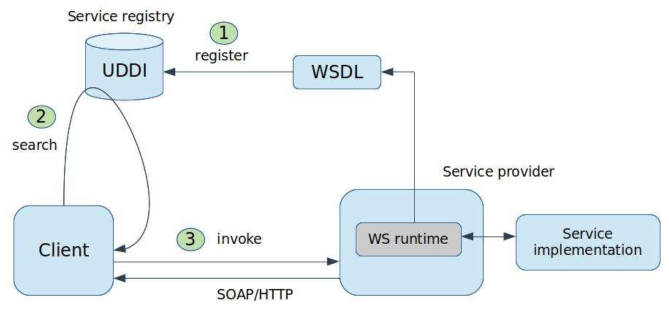
XML-RPC was a precursor of what are web services now.

XML-RPC was a RPC protocol relying of XML for the representation of requests and HTTP for the transport of requests.

The idea was to be independent from execution platforms or languages and to rely on widely recognized and adopted formats.

XML-RPC was a precursor and evolved into SOAP, the protocol used in web services.

SOAP WS : architecture (2000++)



This figure illustrates the architecture of web services (WS).

A service provider may implement a service in any language and/or platform, as soon as a runtime for WS exists in his environment.

The runtime is a composed of

- stub and skeleton generators

- a WSDL generator

- a web server (WS runtime) for making services available on the internet

Then, on the server side, the service implementation is linked with the web server, in order to be able to receive requests through the HTTP communication protocol. A skeleton is generated and is a web application in the web server. A WSDL description (Web Service Description Language) of the service is generated and published, i.e. made available to potential clients.

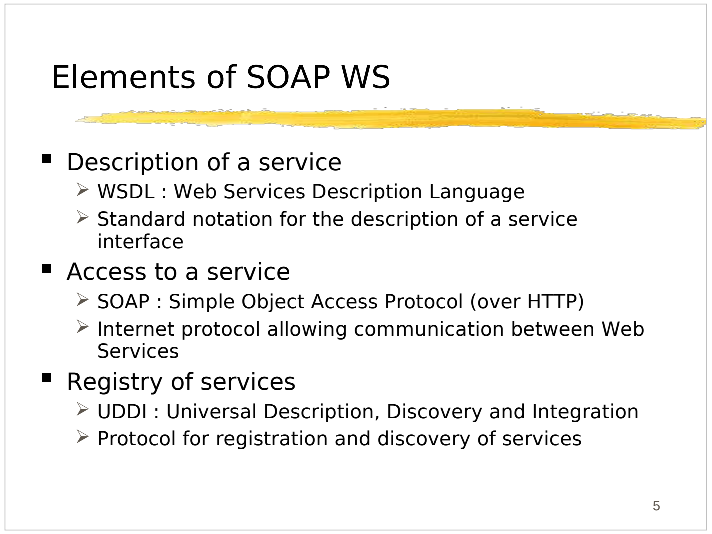
The WS architecture specifies that a service registry (a naming service) should be used for the publication and discovery of WSDL descriptions. However, UDDI was not actually used. Generally the WSDL of a WS can be published on a Web server as any document.

On the client side, the WSDL description can be copied and used to generate a stub in the environment of the client. Notice that the environment of the client is not mandatorily the same as the one of the server. Then the client can implement an application which is able to invoke the WS by calling the stub.

The stub communicates with the skeleton with the SOAP/HTTP protocol which is a standard.

HTTP and SOAP are standards from the W3C.

SOAP describes the syntax of request and response messages which are transported with HTTP.



Elements of SOAP WS

- Description of a service
  - WSDL : Web Services Description Language
  - Standard notation for the description of a service interface
- Access to a service
  - SOAP : Simple Object Access Protocol (over HTTP)
  - Internet protocol allowing communication between Web Services
- Registry of services
  - UDDI : Universal Description, Discovery and Integration
  - Protocol for registration and discovery of services

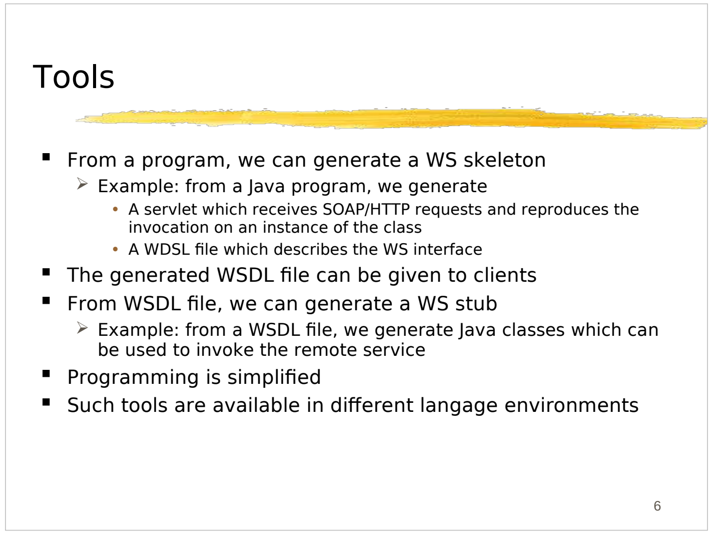
5

Therefore the main elements of WS are :

- the description of the service in WSDL. Generally, from an implementation of a service (e.g. a procedure), tools are provided to generate the WSDL description of the service, which is published for clients. The clients can used this WSDL description to generate stubs so that calls to the service can be programmed easily.

- access protocols which are SOAP (for the content of messages) and HTTP (for the transport). All the WS runtimes (in any environment) comply with these standards.

- registries of service (UDDI) which are not really used.



Tools

- From a program, we can generate a WS skeleton
  - Example: from a Java program, we generate
    - A servlet which receives SOAP/HTTP requests and reproduces the invocation on an instance of the class
    - A WDSL file which describes the WS interface
- The generated WSDL file can be given to clients
- From WSDL file, we can generate a WS stub
  - Example: from a WSDL file, we generate Java classes which can be used to invoke the remote service
- Programming is simplified
- Such tools are available in different langage environments

6

To illustrate this, we give an example of use in the Java environment.

In the Java environment, a WS tool is used to generate from a program (with an exported interface) a skeleton as a servlet. A servlet is a Java program which runs in a web server. This servlet/skeleton received SOAP/HTTP requests and reproduces the invocation on an instance of the class. The WSDL specification of the WS is also generated.

The WSDL file is published on the web and imported by the client.

From the WSDL specification, the client can generate stubs which make it easier to program WS invocations.

In the following slides, we give an example with Apache Axis.

Example: programming a Web Services

- Eclipse JEE
- Apache Axis
- Creation of a Web Service
  - From a Java class
  - In the Tomcat runtime
  - Generation of the WSDL file
- Creation of a client application
  - Generation of stubs from a WSDL file
  - Programming of the client

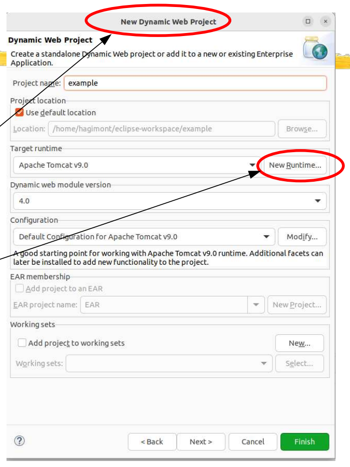
We use Eclipse JEE and Apache Axis which is available in Eclipse JEE.

Apache Axis is used to generate from a Java class a servlet which is installed in the Tomcat engine (the web server). It also generates the WSDL description which describes the interface of the WS.

On the client side, the WSDL description is used to generate stubs which are used to invoke the WS in a client program.

Create a Dynamic Web Project

- Eclipse JEE
- Open JEE perspective
- Create a Dynamic Web Project
- Add your Tomcat runtime


New Dynamic Web Project

Dynamic Web Project

Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name: example

Project location

[✓] Use default location

Location: /home/hagimont/eclipse-workspace/example

Target runtime: Apache Tomcat v9.0

Dynamic web module version: 4.0

Configuration: Default Configuration for Apache Tomcat v9.0

A good starting point for working with Apache Tomcat v9.0 runtime. Additional facets can later be installed to add new functionality to the project.

EAR membership

[ ] Add project to an EAR

EAR project name: EAR

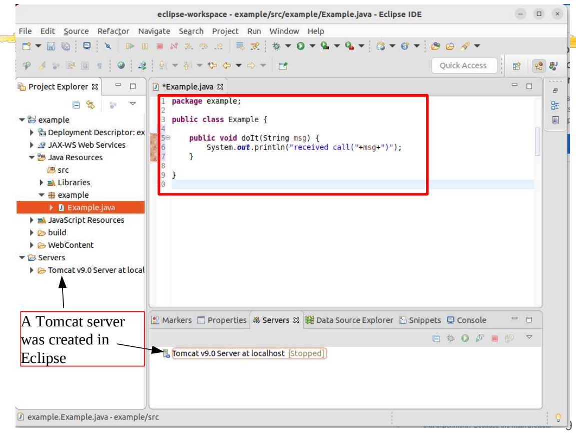
Working sets

[ ] Add project to working sets

Working sets:

In Eclipse, we create a dynamic web project (a project allowing the develop servlets) and add the Tomcat runtime.

Create a Class


eclipse-workspace - example/src/example/Example.java - Eclipse IDE

```java
package example;

public class Example {

    public void doIt(String msg) {
        System.out.println("received call("+msg+")");
    }

}
```

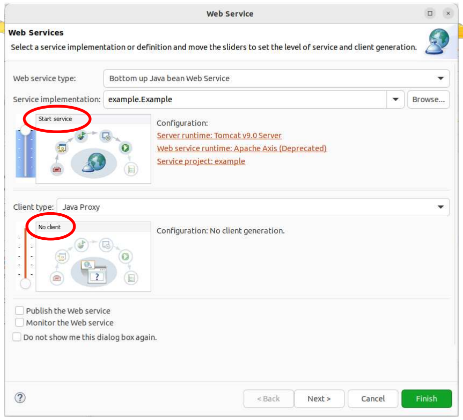
Tomcat v9.0 Server at localhost [Stopped]

A Tomcat server was created in Eclipse

In this project, we create a class.

Notice that a Tomcat server is running in Eclipse.

From source file : Web Service → create Web Service


Web Service

Web Services

Select a service implementation or definition and move the sliders to set the level of service and client generation.

Web service type: Bottom up Java bean Web Service

Service implementation: example.Example

Start service

Configuration:
Server runtime: Tomcat v9.0 Server
Web service runtime: Apache Axis (Deprecated)
Service project: example

Client type: Java Proxy

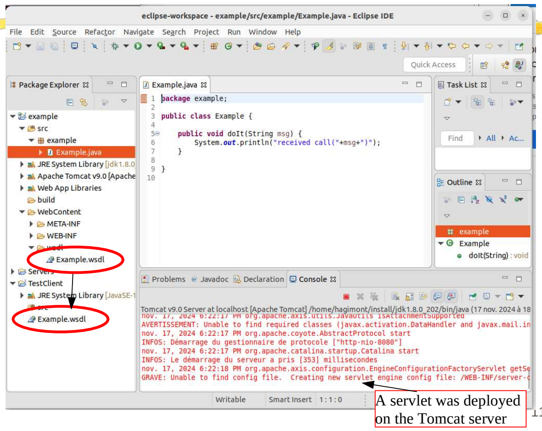
No client

Configuration: No client generation.

- [ ] Publish the Web service
- [ ] Monitor the Web service
- [ ] Do not show me this dialog box again.

From the source file of the class, we can generate (right click) a WS from this file.

Copy the generated WSDL file in a new Java project


eclipse-workspace - example/src/example/Example.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Package Explorer
example
src
example
Example.java
JRE System Library [jdk1.8.0
Apache Tomcat v9.0 [Apache
Web App Libraries
build
WebContent
META-INF
WEB-INF
wsdl
Example.wsdl
Servers
TestClient
JRE System Library [JavaSE-1
src
Example.wsdl
Example.java
Task List
Find All Ac...
Outline
example
Example
doIt(String) : void
Problems Javadoc Declaration Console
Writable Smart Insert 1 : 1 : 0

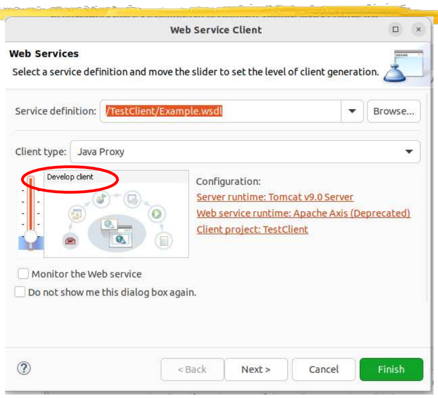
```java
package example;

public class Example {

    public void doIt(String msg) {
        System.out.println("received call("+msg+")");
    }

}
```

```
Tomcat v9.0 Server at localhost [Apache Tomcat] /home/hagimont/install/jdk1.8.0_202/bin/java (17 nov. 2024 à 18
nov. 17, 2024 6:22:17 PM org.apache.axis.utils.JavaUtils isAttachmentSupported
AVERTISSEMENT: Unable to find required classes (javax.activation.DataHandler and javax.mail.in
nov. 17, 2024 6:22:17 PM org.apache.coyote.AbstractProtocol start
INFOS: Démarrage du gestionnaire de protocole ["http-nio-8080"]
nov. 17, 2024 6:22:17 PM org.apache.catalina.startup.Catalina start
INFOS: Le démarrage du serveur a pris [353] millisecondes
nov. 17, 2024 6:22:18 PM org.apache.axis.configuration.EngineConfigurationFactoryServlet getSe
GRAVE: Unable to find config file.  Creating new servlet engine config file: /WEB-INF/server-c
```

A servlet was deployed on the Tomcat server

Then, we create a new Java project and copy the WSDL description in the new project.

From the WSDL file
Web Service → Generate Client (Develop Client)



Web Services

Select a service definition and move the slider to set the level of client generation.

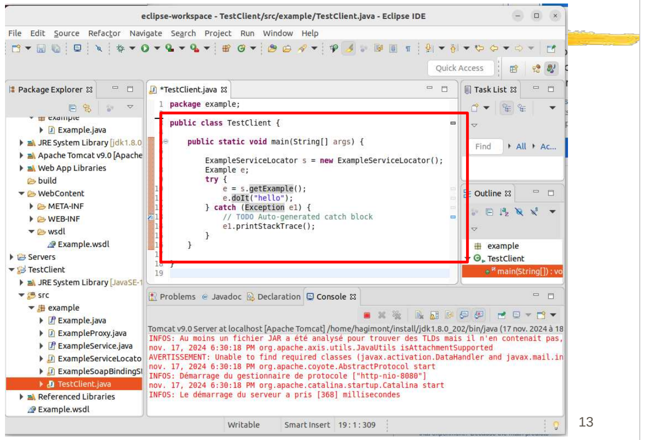
Service definition: /TestClient/Example.wsdl Browse...

Client type: Java Proxy

Configuration:
Server runtime: Tomcat v9.0 Server
Web service runtime: Apache Axis (Deprecated)
Client project: TestClient

In the new project, from the WSDL file, we generate (right click) the stubs (develop Client).

Program a client



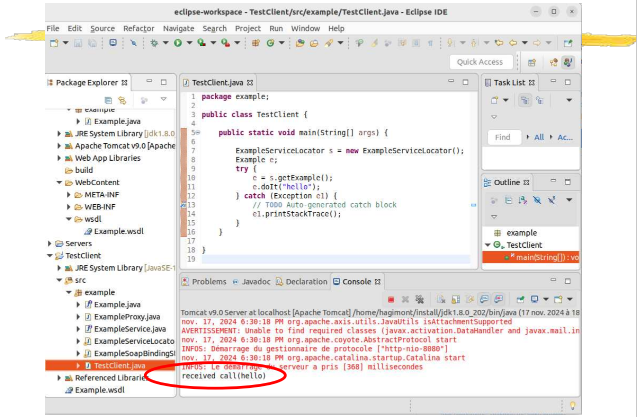
```java
package example;

public class TestClient {

    public static void main(String[] args) {

        ExampleServiceLocator s = new ExampleServiceLocator();
        Example e;
        try {
            e = s.getExample();
            e.doIt("hello");
        } catch (Exception e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        }
    }

}
```

```
Tomcat v9.0 Server at localhost [Apache Tomcat] /home/hagimont/install/jdk1.8.0_202/bin/java (17 nov. 2024 à 18
INFOS: Au moins un fichier JAR a été analysé pour trouver des TLDs mais il n'en contenait pas,
nov. 17, 2024 6:30:18 PM org.apache.axis.utils.JavaUtils isAttachmentSupported
AVERTISSEMENT: Unable to find required classes (javax.activation.DataHandler and javax.mail.in
nov. 17, 2024 6:30:18 PM org.apache.coyote.AbstractProtocol start
INFOS: Démarrage du gestionnaire de protocole ["http-nio-8080"]
nov. 17, 2024 6:30:18 PM org.apache.catalina.startup.Catalina start
INFOS: Le démarrage du serveur a pris [368] millisecondes
```

In the new project, we can program an application which makes an invocation of the WS.

The procedure to follow to invoke the WS depends on the tool used (here Apache Axis).

Run



```java
package example;

public class TestClient {

    public static void main(String[] args) {

        ExampleServiceLocator s = new ExampleServiceLocator();
        Example e;
        try {
            e = s.getExample();
            e.doIt("hello");
        } catch (Exception e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        }
    }

}
```

```
Tomcat v9.0 Server at localhost [Apache Tomcat] /home/hagimont/install/jdk1.8.0_202/bin/java (17 nov. 2024 à 18
nov. 17, 2024 6:30:18 PM org.apache.axis.utils.JavaUtils isAttachmentSupported
AVERTISSEMENT: Unable to find required classes (javax.activation.DataHandler and javax.mail.in
nov. 17, 2024 6:30:18 PM org.apache.coyote.AbstractProtocol start
INFOS: Démarrage du gestionnaire de protocole ["http-nio-8080"]
nov. 17, 2024 6:30:18 PM org.apache.catalina.startup.Catalina start
INFOS: Le démarrage du serveur a pris [368] millisecondes
received call(hello)
```

We can then run the client program which invokes the WS.

Generated WSDL

```
<wsdl:definitions targetNamespace="http://DefaultNamespace"
xmlns:apachesoap="http://xml.apache.org/xml-soap" xmlns:impl="http://DefaultNamespace"
xmlns:intf="http://DefaultNamespace" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!--WSDL created by Apache Axis version: 1.4
Built on Apr 22, 2006 (06:55:48 PDT)-->
 <wsdl:types>
  <schema elementFormDefault="qualified" targetNamespace="http://DefaultNamespace"
xmlns="http://www.w3.org/2001/XMLSchema">
   <element name="doIt">
    <complexType>
     <sequence>
      <element name="msg" type="xsd:string"/>
     </sequence>
    </complexType>
   </element>
   <element name="doItResponse">
    <complexType/>
   </element>
  </schema>
 </wsdl:types>
```

We can have a look at the WSDL description.

We can see that the WSDL syntax is not very simple. Therefore such WSDL descriptions are not written by the user, but generally generated by the tool on the server side and imported by the client.

Generated WSDL

```
<wsdl:message name="doItResponse">
    <wsdl:part element="impl:doItResponse" name="parameters">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="doItRequest">
    <wsdl:part element="impl:doIt" name="parameters">
    </wsdl:part>
  </wsdl:message>
  <wsdl:portType name="MyService">
    <wsdl:operation name="doIt">
       <wsdl:input message="impl:doItRequest" name="doItRequest">
     </wsdl:input>
       <wsdl:output message="impl:doItResponse" name="doItResponse">
     </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
```

Very verbose !

Generated WSDL

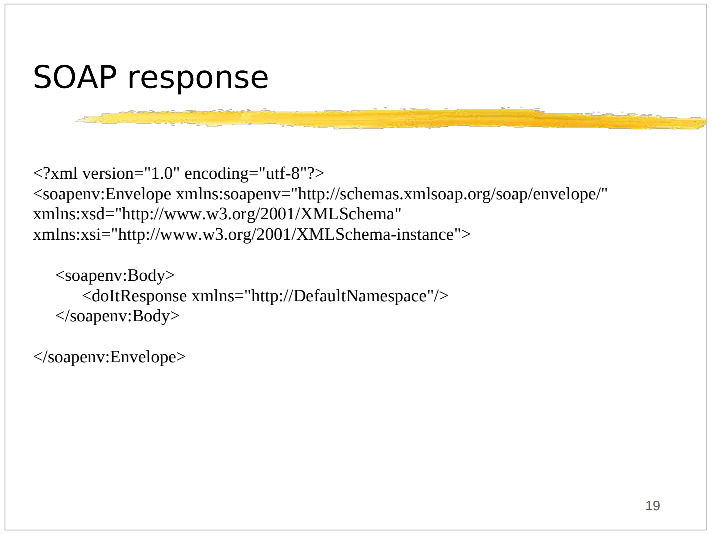
```
<wsdl:binding name="MyServiceSoapBinding" type="impl:MyService">
    <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="doIt">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="doItRequest">
        <wsdlsoap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="doItResponse">
        <wsdlsoap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="MyServiceService">
    <wsdl:port binding="impl:MyServiceSoapBinding" name="MyService">
      <wsdlsoap:address location="http://localhost:8080/HW/services/MyService"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Very very verbose !

SOAP request(with TCP/IP Monitor)

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <soapenv:Body>
    <doIt xmlns="http://DefaultNamespace">
      <msg>hello</msg>
    </doIt>
  </soapenv:Body>

</soapenv:Envelope>
```

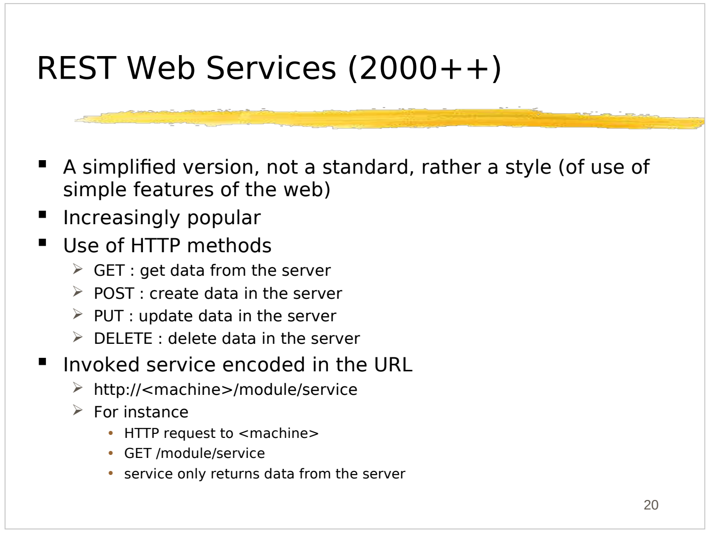
We can have a look at the SOAP request. This is simply a standardized format for exchanged messages.



SOAP response

```
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

   <soapenv:Body>
      <doItResponse xmlns="http://DefaultNamespace"/>
   </soapenv:Body>

</soapenv:Envelope>
```

Here is the SOAP response.



REST Web Services (2000++)

- A simplified version, not a standard, rather a style (of use of simple features of the web)
- Increasingly popular
- Use of HTTP methods
  - GET : get data from the server
  - POST : create data in the server
  - PUT : update data in the server
  - DELETE : delete data in the server
- Invoked service encoded in the URL
  - http://<machine>/module/service
  - For instance
    - HTTP request to <machine>
    - GET /module/service
    - service only returns data from the server

SOAP/WSDL based WS were very popular some years ago. They are becoming obsolete.

An evolution of WS is REST WS. This is a simplified version which is very popular now. Notice that REST WS is not a standard, but rather a recommendation or a style of implementation, based on simple features of the web.

It relies on HTTP requests (GET, POST, PUT, DELETE), but mainly GET and POST are used. GET is used when you want to read (only) data from the WS while POST is used when you want to modify something in the WS.

The service that you call is encoded in the URL.

For instance a GET HTTP request on URL http://<machine>/module/service

This service is supposed to return data from the server, without updating anything in the server (its a style, it is not enforced).

REST Web Services

- Parameter passing
  - HTTP parameters
    - Format : field1=value1&field2=value2
    - GET : parameters in the URL
      - http://nom_du_serveur/cgi-bin/script.cgi?champ1=valeur1&...
    - POST : parameters are included in the body of the HTTP request
  - XML or JSON (or any other)
    - Included in the body of the HTTP request
- Response
  - Included in the body of the HTTP response: XML or JSON (or any other)
- Description of a REST WS (available on the net)
  - A simple document describing the methods and parameters
  - No WSDL

Parameter passing can be based on HTTP parameters. With HTTP parameters, parameters are encoded as a String field1=value1&field2=value2 …

This parameter string is passed in the URL if you use the GET HTTP method, and it is passed in the body of the request if you use the POST HTTP method.

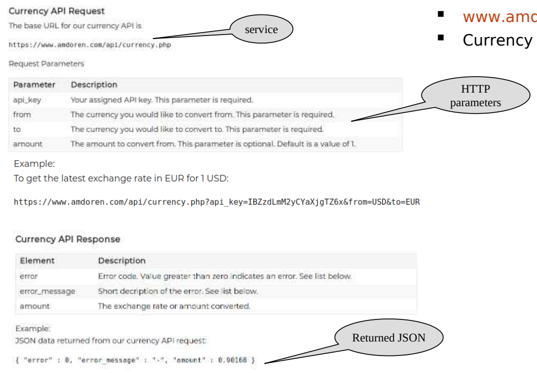
You can also pass parameters in a document (XML or JSON or any format) in the body of the request.

A service can return a document (XML or JSON) in the body of the response (which corresponds to return parameters).

The description of a REST WS is simply a document describing the services that you may call and the passed parameters (names, formats).

Example of existing REST WS

- www.amdoren.com
- Currency converter



Currency API Request

The base URL for our currency API is

```
https://www.amdoren.com/api/currency.php
```

Request Parameters

Parameter	Description
api_key	Your assigned API key. This parameter is required.
from	The currency you would like to convert from. This parameter is required.
to	The currency you would like to convert to. This parameter is required.
amount	The amount to convert from. This parameter is optional. Default is a value of 1.

Example:
To get the latest exchange rate in EUR for 1 USD:

```
https://www.amdoren.com/api/currency.php?api_key=IBZzdLmM2yCYaXjgTZ6x&from=USD&to=EUR
```

Currency API Response

Element	Description
error	Error code. Value greater than zero indicates an error. See list below.
error_message	Short decription of the error. See list below.
amount	The exchange rate or amount converted.

Example:
JSON data returned from our currency API request:

```
{ "error" : 0, "error_message" : "-", "amount" : 0.90168 }
```

Here is an example of description of a REST WS. This is for a currency converter.

It says that you have one service available :

https://www.amdoren.com/api/currency.php

It lists the parameters that may be passed in the HTTP GET request. A example is given.

It then describes the response which is a JSON. A example is given.

Development of REST WS

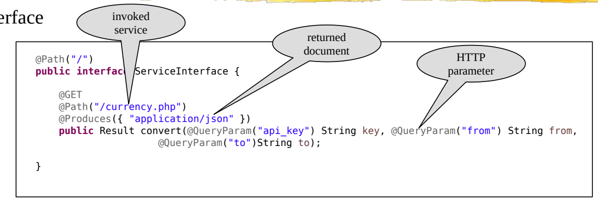
- You can develop your application by hand in any programming langage
  - Verbose and error prone
  - As for RPC, code can be generated
- Many development environments
  - e.g. Resteasy and Jersey
  - Resteasy in the following of the talk (client and server sides)
  - A view on Spring (mainly server side)

Many development environments can be used to REST WS development.

In the following, we overview the used of resteasy (on the server side and the client side) and we have a look at Spring. Both will be used in the labs.

Existing REST WS (client with resteasy)

Interface



```
@Path("/")
public interface ServiceInterface {

    @GET
    @Path("/currency.php")
    @Produces({ "application/json" })
    public Result convert(@QueryParam("api_key") String key, @QueryParam("from") String from,
                     @QueryParam("to")String to);

}
```

Java bean (generated from JSON)

```
public class Result {

    String error;
    String error_message;
    String amount;

    // getters/setters
```

With the currency converter, as said in the documentation, the conversion method takes 3 HTTP parameters (api_key, from, to, the last is optional) and it returns a JSON.

The 3 HTTP parameters are associated with Java parameters (with @QueryParam) and a Java bean is created for the JSON.

Existing REST WS (client with resteasy)

Client

```
public class Client {

    public static void main(String args[]) {

        final String path = "https://www.amdoren.com/api";

        ResteasyClient client = new ResteasyClientBuilder().build();
        ResteasyWebTarget target = client.target(UriBuilder.fromPath(path));
        ServiceInterface proxy = target.proxy(ServiceInterface.class);

        Result r = proxy.convert("9xwjRjxTtnzuGKH7LcWC5Vengr52F3", "EUR", "AMD");

        System.out.println("convert: "+r.getError()+"/"+r.getError_message()
                                        +"/"+r.getAmount()) ;
    }
}
```

easy
invocation

25

And here is an example of client which invokes the service.

Implementing a service with resteasy

- WS class

```
 @Path("/")
public class Facade {

    static Hashtable<String, Person> ht = new Hashtable<String, Person>();

    @POST
    @Path("/addperson")
    @Consumes({ "application/json" })
    public String addPerson(Person p) {
         ht.put(p.getId(), p);
         return "person added";
    }

    @GET
    @Path("/getperson")
    @Produces({ "application/json" })
    public Person getPerson(@QueryParam("id") String id) {
         return ht.get(id);
    }

    @GET
    @Path("/listpersons")
    @Produces({ "application/json" })
    public Collection<Person> listPersons() {
         return ht.values();
    }
}
```

Receives a JSON
Deserialized into a Java object
Returns a String

Returns an object
Serialized into a JSON
Receives an id HTTP parameter

Person is a simple POJO

26

As for SOAP/WS, many tools were implemented to help developers.

Here, we present Resteasy (Jersey is also a very popular one you may look at).

On the server side, you can use annotations in a Java program to say :

- each method is associated with a path in the URL used to access the WS

- @Path : specifies the element of the path associated with the class or the method. Here method addPerson() is associated with path /addperson

- @POST or @GET : specifies which HTTP method is used. Notice that GET returns an object (data) while POST returns an HTTP code (and a message).

- @Consumes : specifies that we receive a JSON object which is deserialized into a Java object.

- @Produces : specifies that we return a Java object which is serialized into a JSON object.

- @QueryParam : the getPerson() method has an "id" parameter. The QueryParam annotation associates this parameter with an "id" HTTP parameter.

Implementing a service with resteasy

- Add the RestEasy jars in Tomcat (lib)
- In eclipse (not easy with vscode)
  - Create a Dynamic Web Project
  - Add RestEasy jars in the buildpath
  - Create a package
  - Implement the WS classes (Facade + Person)
  - Add a class RestApp

```java
public class RestApp extends Application {
	private Set<Object> singletons = new HashSet<Object>();
	public RestApp() {
		singletons.add(new Facade());
	}
	public Set<Object> getSingletons() {
		return singletons;
	}
}
```

To run this example :

- add the Resteasy jars in Tomcat and Eclipse
- create a dynamic web project (a servlet project)
- add the RestEasy jars in the buildpath of the project
- create a package and the previously developed classes
- add the RestApp class

Implementing a service with resteasy

- Add a web.xml descriptor in the WebContent/WEB-INF folder

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd" version="3.1">
  <display-name>essai-server</display-name>
  <servlet>
    <servlet-name>resteasy-servlet</servlet-name>
    <servlet-class>
            org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher
        </servlet-class>
    <init-param>
      <param-name>javax.ws.rs.Application</param-name>
      <param-value>pack.RestApp</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>resteasy-servlet</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

- Export the war in Tomcat

Add the web.xml descriptor in the WebContent/WEB-INF folder and export a war (in the webapps folder of Tomcat)

Publish the WS

- Just write a documentation which says that
  - The WS is available at `http://<machine-name>:8080/<project-name>/`
  - Method addperson with POST receives a person JSON :

```
{
  "id":"00000",
  "firstname":"Alain",
  "lastname":"Tchana",
  "phone":"0102030405",
  "email":"alain.tchana@enseeiht.fr"
}
```

  - Method getperson with GET receives an HTTP parameter id and returns a person JSON
  - Method listperson returns a JSON including a set of persons
- A caller may use any tool (not only RestEasy)

Publication of a REST WS is simply a document describing the interface.

Implementing the client with resteasy

- From the previous documentation, a client can write the interface

```
@Path("/")
public interface FacadeInterface {

    @POST
    @Path("/addperson")
    @Consumes({ "application/json" })
    public String addPerson(Person p);

    @GET
    @Path("/getperson")
    @Produces({ "application/json" })
    public Person getPerson(@QueryParam("id") String id);

    @GET
    @Path("/listpersons")
    @Produces({ "application/json" })
    public Collection<Person> listPersons();
}
```

On the client side, from the documentation, a user can write a Java interface with Resteasy annotations. Of course, it's very similar to what we wrote on the server side, but we could do it for a WS we don't know (we only have the documentation).

Implementing the client with resteasy

- And write a class which invokes the WS

```
public class TestClient {

    public static void main(String[] args) {

        final String path = "http://localhost:8080/rest-server";

        ResteasyClient client = new ResteasyClientBuilder().build();
        ResteasyWebTarget target = client.target(UriBuilder.fromPath(path));
        FacadeInterface proxy = target.proxy(FacadeInterface.class);

        String resp;
        resp = proxy.addPerson(new Person("007","James Bond"));
        System.out.println(resp);
        resp = proxy.addPerson(new Person("006","Dan Hagi"));
        System.out.println(resp);

        System.out.print("list Person: ");
        Collection<Person> l = proxy.listPersons();
        for (Person p : l) System.out.print("["+p.getId()+"/"+p.getName()+"]");
        System.out.println();

        Person p = proxy.getPerson("006");
        System.out.println("get Person: "+p.getId()+"/"+p.getName());
    }
}
```

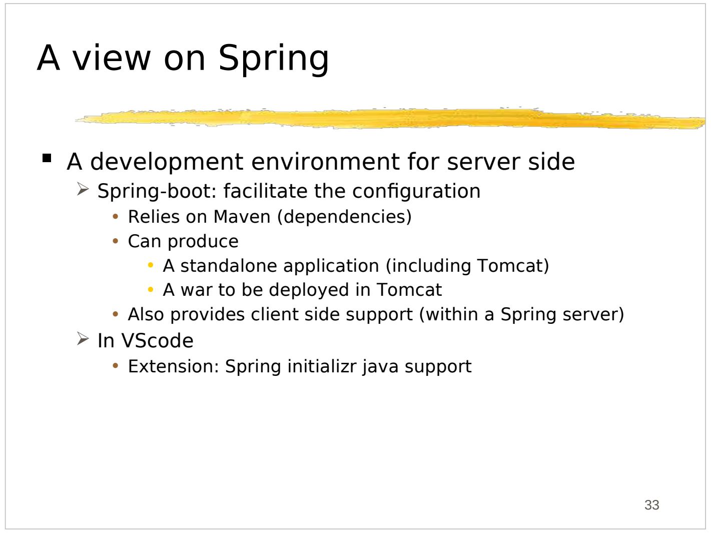
The previous annotated interface (FacadeInterface) makes it easy to invoke the service. We can build a proxy object of type FacadeInterface.

This proxy allows programming service invocations simply as method calls.

Implementing the client with resteasy

- In eclipse or vscode
  - Create a Java Project
  - Add RestEasy jars in the project
  - Implement the Java bean that correspond to the JSON
    - Automatic generation with https://www.site24x7.com/tools/json-to-java.html
  - Implement the interface and the client class (FacadeInterface + TestClient)
  - Run

This is the procedure to run the client.



A view on Spring

- A development environment for server side
  - Spring-boot: facilitate the configuration
    - Relies on Maven (dependencies)
    - Can produce
      - A standalone application (including Tomcat)
      - A war to be deployed in Tomcat
    - Also provides client side support (within a Spring server)
  - In VScode
    - Extension: Spring initializr java support

Spring is a development environment for developing REST WS.

Spring-boot is an extension which simplifies the configuration. It relies on Maven.

Thanks to Spring-boot, you can produce:

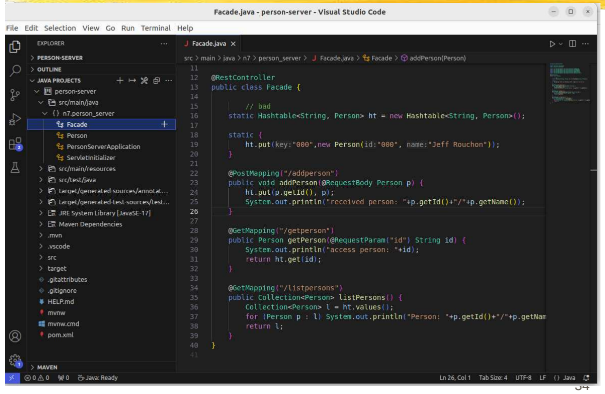
- an application which embeds Tomcat. When you launch it, you start a Tomcat web server which included your REST WS application.

- a war archive which includes your REST WS application. This war can be deployed in a running Tomcat server.

Spring also provides support for invoking an external REST WS from a Spring WS.

In VScode, you can use the "Spring initializr" extension which automates the creation of a Spring-boot project.

Spring initializ : Create a Maven Project
Implement an annotated class


Facade.java - person-server - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
PERSON-SERVER
OUTLINE
JAVA PROJECTS
person-server
src/main/java
n7.person_server
Facade
Person
PersonServerApplication
ServletInitializer
src/main/resources
src/test/java
target/generated-sources/annotat...
target/generated-test-sources/test...
JRE System Library [JavaSE-17]
Maven Dependencies
.mvn
.vscode
src
target
.gitattributes
.gitignore
HELP.md
mvnw
mvnw.cmd
pom.xml
MAVEN
Facade.java
src > main > java > n7 > person_server > Facade.java > Facade > addPerson(Person)
Ln 26, Col 1 Tab Size: 4 UTF-8 LF {} Java

```java

@RestController
public class Facade {

        // bad
    static Hashtable<String, Person> ht = new Hashtable<String, Person>();

    static {
        ht.put(key:"000",new Person(id:"000", name:"Jeff Rouchon"));
    }

    @PostMapping("/addperson")
    public void addPerson(@RequestBody Person p) {
        ht.put(p.getId(), p);
        System.out.println("received person: "+p.getId()+"/"+p.getName());
    }

    @GetMapping("/getperson")
    public Person getPerson(@RequestParam("id") String id) {
        System.out.println("access person: "+id);
        return ht.get(id);
    }

    @GetMapping("/listpersons")
    public Collection<Person> listPersons() {
        Collection<Person> l = ht.values();
        for (Person p : l) System.out.println("Person: "+p.getId()+"/"+p.getNam
        return l;
    }
}

```

Assuming you have installed in VScode the Spring initializr java support extension.

Here, we create the same REST WS as with RestEasy (person management).

You can create a Spring initializr project.

Initially, there are 2 classes in the project (PersonServerApplication and ServletInitializer). You don't have to modify them.

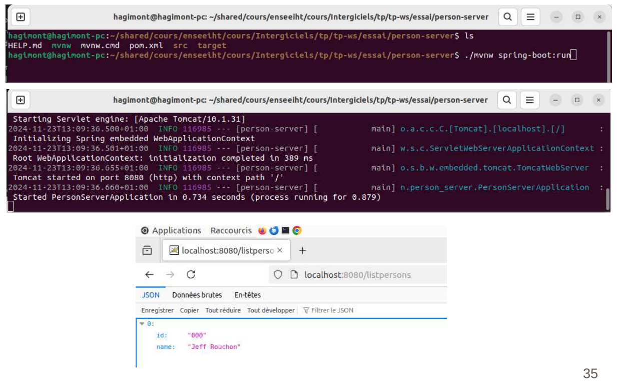
We just implement a class (Facade) which implements the methods of the REST WS.

You can annotate these methods with @GetMapping and @PostMapping.

Method parameters can be annotated with @RequestParam (for HTTP parameters) of @RequestBody (for a JSON object).

By default, the returned object is serialized into a JSON object.

Run as a standalone application
(maven is used to run the app, including Tomcat)



```
hagimont@hagimont-pc:~/shared/cours/enseeiht/cours/Intergiciels/tp/tp-ws/essai/person-server$ ls
HELP.md  mvnw  mvnw.cmd  pom.xml  src  target
hagimont@hagimont-pc:~/shared/cours/enseeiht/cours/Intergiciels/tp/tp-ws/essai/person-server$ ./mvnw spring-boot:run
```

```
 Starting Servlet engine: [Apache Tomcat/10.1.31]
2024-11-23T13:09:36.500+01:00  INFO 116985 --- [person-server] [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       :
 Initializing Spring embedded WebApplicationContext
2024-11-23T13:09:36.501+01:00  INFO 116985 --- [person-server] [           main] w.s.c.ServletWebServerApplicationContext :
 Root WebApplicationContext: initialization completed in 389 ms
2024-11-23T13:09:36.655+01:00  INFO 116985 --- [person-server] [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  :
 Tomcat started on port 8080 (http) with context path '/'
2024-11-23T13:09:36.660+01:00  INFO 116985 --- [person-server] [           main] n.person_server.PersonServerApplication  :
 Started PersonServerApplication in 0.734 seconds (process running for 0.879)
```

localhost:8080/listpersons

```
▼ 0:
    id:    "000"
    name:  "Jeff Rouchon"
```

In you VScode project, you have a "mvnw" script. It's a Maven script.

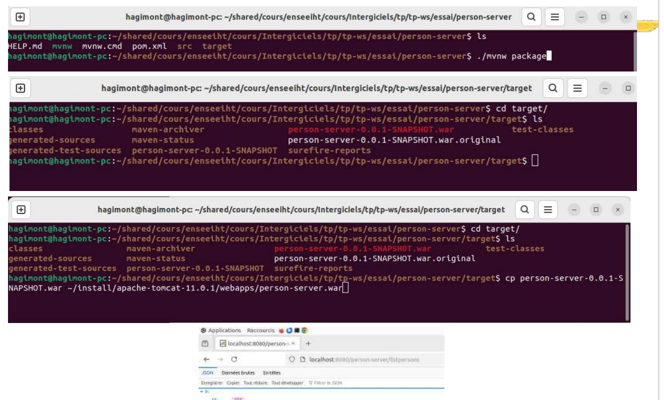
If you run: ./mvnw spring-boot:run

It will start a Tomcat server which executes your REST WS. Thanks to Maven, it downloads all the dependencies.

You can test your WS with a web bowser.

Notice that the URL (http://localhost:8080/listpersons) used in the bowser only includes the invoked method name.

Export a war in Tomcat



```
hagimont@hagimont-pc:~/shared/cours/enseeiht/cours/Intergiciels/tp/tp-ws/essai/person-server$ ls
HELP.md  mvnw  mvnw.cmd  pom.xml  src  target
hagimont@hagimont-pc:~/shared/cours/enseeiht/cours/Intergiciels/tp/tp-ws/essai/person-server$ ./mvnw package
```

```
hagimont@hagimont-pc:~/shared/cours/enseeiht/cours/Intergiciels/tp/tp-ws/essai/person-server$ cd target/
hagimont@hagimont-pc:~/shared/cours/enseeiht/cours/Intergiciels/tp/tp-ws/essai/person-server/target$ ls
classes                  maven-archiver                person-server-0.0.1-SNAPSHOT.war           test-classes
generated-sources        maven-status                  person-server-0.0.1-SNAPSHOT.war.original
generated-test-sources   person-server-0.0.1-SNAPSHOT  surefire-reports
hagimont@hagimont-pc:~/shared/cours/enseeiht/cours/Intergiciels/tp/tp-ws/essai/person-server/target$
```

```
hagimont@hagimont-pc:~/shared/cours/enseeiht/cours/Intergiciels/tp/tp-ws/essai/person-server$ cd target/
hagimont@hagimont-pc:~/shared/cours/enseeiht/cours/Intergiciels/tp/tp-ws/essai/person-server/target$ ls
classes                  maven-archiver                person-server-0.0.1-SNAPSHOT.war           test-classes
generated-sources        maven-status                  person-server-0.0.1-SNAPSHOT.war.original
generated-test-sources   person-server-0.0.1-SNAPSHOT  surefire-reports
hagimont@hagimont-pc:~/shared/cours/enseeiht/cours/Intergiciels/tp/tp-ws/essai/person-server/target$ cp person-server-0.0.1-S
NAPSHOT.war ~/install/apache-tomcat-11.0.1/webapps/person-server.war
```

localhost:8080/person-server/listpersons

```
▼ 0:
    id:    "000"
    name:  "Jeff Rouchon"
```

Another way to run your REST WS is to export a war archive that you deploy in a Tomcat server (already running).

In your VScode project, you can run: ./mvnw package

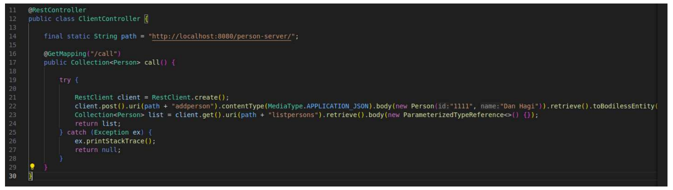
It will produce a war in the target folder (person-server-0.0.1-SNAPSHOT.war).

Here, I copy this war into my Tomcat (webapps folder).

I rename it as person-server.war as the name of the war is used in the URL used to access the WS.

The URL (http://localhost:8080/person-server/listpersons) used in the bowser includes the name of the war.

Invoking a rest API from Spring



```java
@RestController
public class ClientController {

    final static String path = "http://localhost:8080/person-server/";

    @GetMapping("/call")
    public Collection<Person> call() {

        try {

            RestClient client = RestClient.create();
            client.post().uri(path + "addperson").contentType(MediaType.APPLICATION_JSON).body(new Person(id:"1111", name:"Dan Hagi")).retrieve().toBodilessEntity(
            Collection<Person> list = client.get().uri(path + "listpersons").retrieve().body(new ParameterizedTypeReference<>() {});
            return list;
        } catch (Exception ex) {
            ex.printStackTrace();
            return null;
        }
    }
}
```

Spring also provides support for invoking external REST WS from withing a Spring WS.

The class to use for that is RestClient.

Here, I programmed a simple Spring REST WS with a method "call". In that method, I invoke methods "addperson" and "listpersons" from the previous service.

Conclusion

- Web Services: a RPC over HTTP, exchanging XML or JSON
- Interesting for heterogeneity as there are tools in all environments
- Recently
  - SOAP WS less used
  - REST + XML/JSON more popular
  - Micro-services: used for structuring backend applications

To conclude, Web services aim at implementing a RPC service on top of HTTP and relying on standard formats (XML, JSON).

One of the main interest is the independence between the server (the service provider) and the client (the service consumer). They can be from different organizations and use different tools, OS, or languages.

The recent evolution is an obsolescence of SOAP and an increased popularity of REST and JSON.

Recently, the micro-service architecture was proposed. It consists in architecturing large applications (especially backend applications) in terms of a set of interconnected REST WS (components). The advantage is indenpendence between components.