Every Ware Lab

UNIVERSITÀ DEGLI STUDI DI MILANO

REST Servers and MQTT

Luca Arrotta and Riccardo Presotto

EWLab – Università degli studi di Milano

Professor: Claudio Bettini

These slides are based on previous versions created by Letizia Berolaja, Sergio Mascetti, Dario Freni, Claudio Bettini and Gabriele Civitarese

Copyright

Some slides for this course are partly adapted from the ones distributed by the publisher of the reference book for this course (Distributed Systems: Principles and Paradigms, A. S. Tanenbaum, M. Van Steen, Prentice Hall, 2007).

All the other slides are from the teacher of this course. All the material is subject to copyright and cannot be redistributed without consent of the copyright holder. The same holds for audio and video-recordings of the classes of this course.

REST (Representational State Transfer)

- Machine-to-machine communication via HTTP
- REST architecture relies over four HTTP methods: **Get, Post, Put, Delete**
- It was introduced in 2000 and it is currently used by Yahoo, Google, Instagram, Twitter etc... To realize many services
- The main advantage with respect to Web service **SOAP** and Web service **WSDL-based**: **Simplicity**

Design Principles

1. Explicit use of HTTP methods
2. Be “Stateless”
3. Each resource is identified by a URI (uniform resource identifier) and the structure of these resources is similar to the one of directories.
2. Use of XML and/or Json

Using explicit HTTP methods: errors

- When defining server operations:
  - Use GET always
  - Define with a parameter which operation I want to perform

    Example: http://myservice.it?action=adduser&name=gianmario

- **Problems:**
  - Semantics: in HTTP the GET method is defined to obtain information, not to insert them
  - Unintentional modification of server data, for example by a crawler.

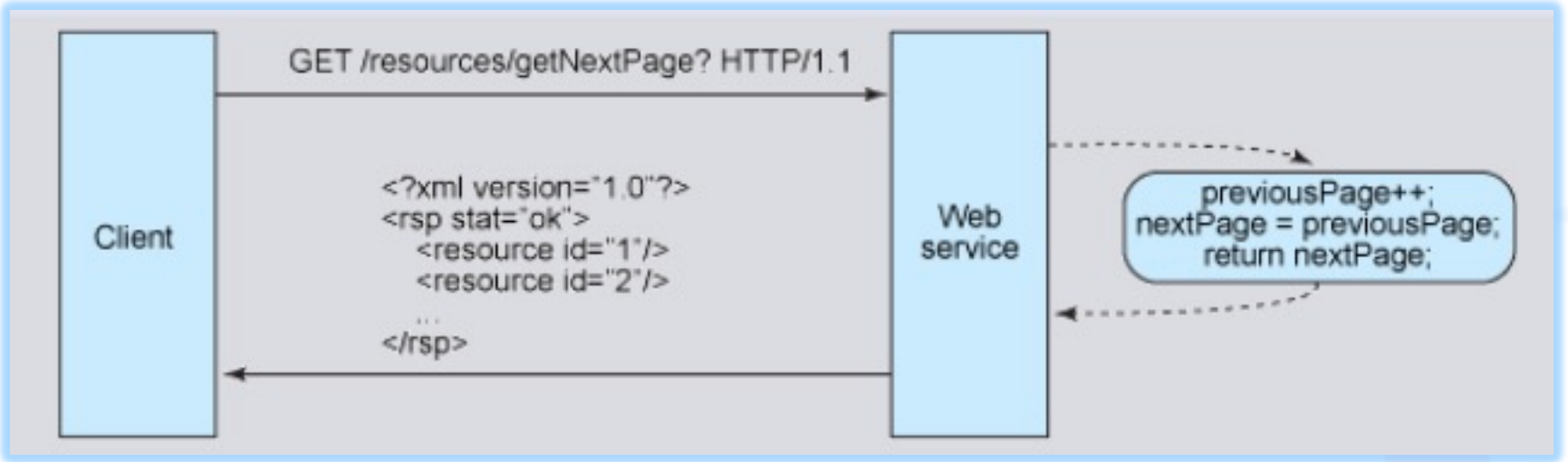
Solution: Using explicit HTTP methods

- REST requires the explicit use of HTTP methods
  - One-to-one **correspondence** between **CRUD** operations (create, read, update, and delete) and **HTTP** methods

CRUD	**<->**	**HTTP**
Create	**<->**	POST
Read	**<->**	GET
Update	**<->**	PUT
Delete	**<->**	DELETE

Every Ware Lab

Stateful services problem



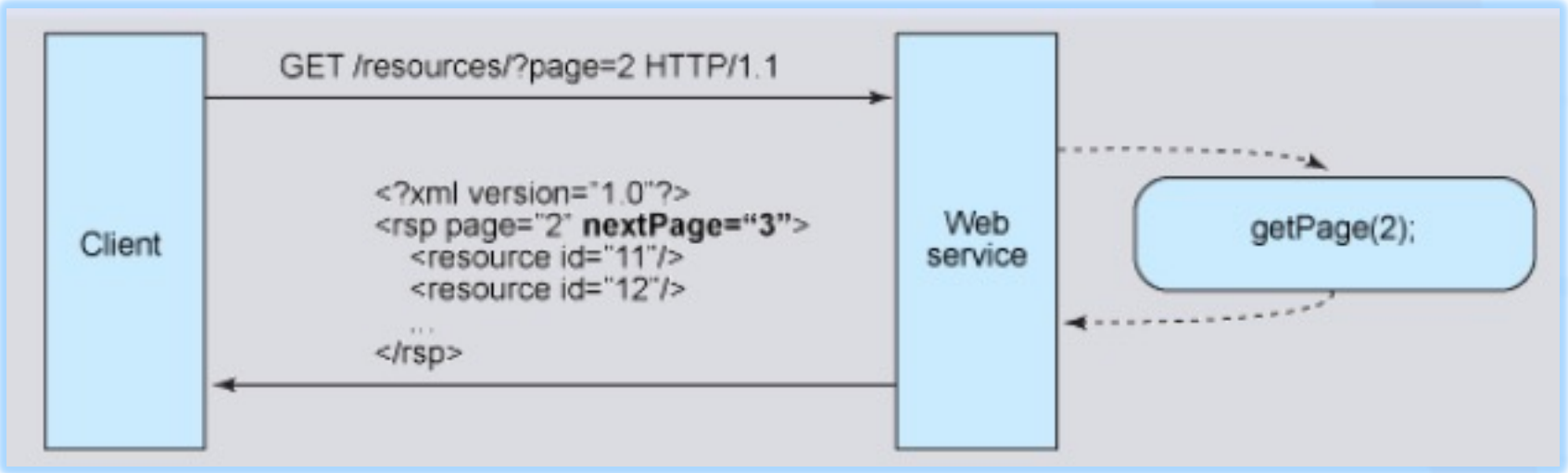
Source: RESTful Web services: The basics

- **Problems:**
  - The server that responds to the client's requests must always be **the same** because it has **to store the client's state**
  - This fact limits the system **scalability, load-balancing**, and **failover**

Every Ware Lab

Solution: Stateless Approach

- Stateless server principles:
  - Each **client request** is **complete** and **independent** The HTTP header and body contain **all the parameters** necessary for the server to perform the requested operation.
  - The **server** does **not** have to **keep** the client **state**.



Source: RESTful Web services: The basics

Every Ware Lab

Server REST resources

- **Resources are the crucial** elements of REST servers
  - CRUD operations refer to resources
  - In each operation the client transmits or receives a representation of a resource

- **Organization** importance
  - The **simplicity** of interacting with a REST server strongly **depends on** how the **resources** are **organized**
  - It is proper to follow some **representation conventions** that help to:
    - Define the client that interacts with the REST server.
    - Limit errors.

Resource Organization Conventions

- **“Predictable” tree structure**. Example:

  http://www.repubblica.it/tecnologia/2013/05/01/foto/festa_dei_lavoratori_l_omaggio_di_google-57826702/1/

- **Avoid** reporting **server-side implementation technology (ex: .php**) so that you can change it
- Keep all **lower-case**
- **Replace spaces** with "-" or ”_"

Every Ware Lab

Which technologies will we use?

- **Jersey**, as a library to define which methods to call relying on:
  - HTTP verbs
  - URI

- **JAXB**, for automatic **marshaling** and **unmarshalling** in JSON or XML

Every Ware Lab

Jersey

- Open-Source Library for Web Service REST Development
- Exploits the JAX-RS API
- Integrates with various servlets (Tomcat, GlassFish,...)
- Enables to map HTTP requests to Java methods

```
@Path("/hello")
public class Hello {
        @GET
        @Produces(MediaType.TEXT_PLAIN)
        public String sayPlainTextHello(){
          return "Hello, world";
    }
}
```

Every Ware Lab

Key Features

- **Annotations** are used to define the **interaction** between **Jersey** and **our code.**
- **One** or **more classes** manage the **actions** (ex: GET)
- **Each class manages all actions** for a specific **level of the URI tree**
- The real **resources are objects** in the central memory (in appropriate data structures) or data on DB

Every Ware Lab

URI composition

- Each **URI** consists of **two parts**:
  - A **prefix** defined at the configuration level
    - For example: http://localhost:8080/Rubrica/rest
  - A **suffix** that changes according to the resources on which we want to apply the actions
    - For example: /user/Giandavide

Every Ware Lab

JAXB

- Additional specification for **marshaling** and **unmarshalling** of Java objects (JavaBeans only) in XML / JSON

- Integrated in Jersey

- Use of annotations for:
  - Define the XML root (@XMLRootElement)
  - Specify variable types (if different from those used in Java)
  - The names to be assigned to the elements (If they differ from those specified in the code)

JAXB (2)

```
@XmlRootElement
public class Word {
  private String name, definition;
  public Word() { }

  public String getName() {return name;}
  public void setName(String name) {this.name = name;}
  public String getDefinition() {return definition;}
  public void setDefinition(String definition) {this.definition = definition;}
}
```

A simple Tutorial: http://www.vogella.com/tutorials/JAXB/article.html

HTTP Verbs

- To **match** the execution of a **method** to an **HTTP verb** it is necessary to use annotations:
  - @GET
  - @POST
  - @PUT
  - @DELETE

```
@GET
public String sayHello() {
    return "Hello, world";
}
```

@Path Annotation

- Can be used for a **class** and / or for **methods**
- Can contain both **variable** and **constant** values
- The **variable** parts are called ***path templates***

```
@Path("/dictionary") //path costante
public class DictonaryResource {

@GET
@Path("{word}") //template
@Produces({"application/xml", "application/json"})
```

Every Ware Lab

Other Annotations

@Path Param

Allows to associate templates to variables

```
...
@GET
@Path("{word}") //template
@Produces({"application/xml", "application/json"})
public Response getWord(@PathParam("word") String name){
...
```

Every Ware Lab

Other Annotations (2)

@Produces

Allows specifying the format returned by a method (header Accept in HTTP)

```
...
@GET
@Produces(MediaType.TEXT_PLAIN)
public String sayPlainTextHello() {
    return "Hello, world";
}
@GET
@Produces(MediaType.TEXT_XML)
public String sayXMLHello() {
   return "<?xml version=\"1.0\"?>" + "<hello> Hello, World! </Hello>";
}
@GET
@Produces(MediaType.TEXT_HTML)
                ...
```

Every Ware Lab

Other Annotations (3)

@Consume

Allows specifying in which format the data is expected from the client

```
...
@POST
@Path("/user")
@Consumes({"application/xml", "application/json"})
public Response put(User userElement) {
    users.addUser(u);
    return Response.ok().build();
    }
...
```

Every Ware Lab

Response

- The Response class is used to generate the HTTP response using **standard codes** *(200 for OK, 404 not found,...)*

```
//risposta 200
return Response.ok().build();
//risposta 200 con annesso oggetto serializzato (via JAXB, in base a Produces)
return Response.ok(oggetto).build();
//risposta 404
return Response.status(Response.Status.NOT_FOUND).build();
```

Every Ware Lab

Exercise: Rest Dictionary

- Create a REST service that allows managing a words dictionary. It should enable the user to:
  - Enter a word and its definition
  - Change the definition of a word
  - Given a word, view its definition.
  - Delete a word
- Manage errors with appropriate HTTP responses (word already entered, the word does not exist,...)
- Pay attention to **synchronization problems!**
- One way to test a REST server on the fly is by using tools like Advanced REST Client (plugin for Chrome)

References

- RESTful Web services: The basics
- REST- Fielding dissertation, Chap. 5

Every Ware Lab

MQTT (MQ Telemetry Transport)

- Machine-to-machine communication via HTTP
- IMB's Andy Stanford Clark and Eurotech's Arlen Nipper created the protocol in 1999.
- OASIS standard. The specification is managed by the OASIS MQTT Technical Committee (http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html)

Every Ware Lab

Design Principles

- Based on **Publish-subscriber** pattern
  - **Publishers** and **subscribers** never contact each other directly.
  - A **broker filters** all incoming messages from publishers and **distributes** them to subscribers.

- **Scalability**
  - Operations on the broker can be highly parallelised and **messages** can be **processed** in an **event-driven way**

- **Asynchronicity**
  - Processes are not blocked while waiting for a message or publishing a message

MQTT client and broker

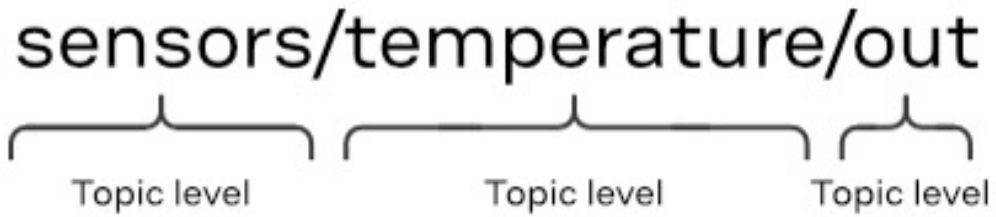
Don't think, "client and server", think, "client and broker" instead.

- MQTT **Client**
  - Any *Thing* connected to the internet (from microcontrollers to a massive server) can effectively become a MQTT client.
  - Both publisher and subscriber are MQTT clients.

- MQTT **Broker**
  - Handles authentication, connections, sessions, and subscriptions
  - Its main responsibility is to receive messages from publishers and **forward** them to subscribers

MQTT Topics

- MQTT messages are published to "topics".
  - Topics consist of one or more topic levels, separated by a forward slash:



  - Topics are case sensitive
  - Topics don’t have to be pre-registered at the broker
  - Topics are a great way to organise the data flows through the network.

Every Ware Lab

MQTT Topics (2)

- Scaling example:
  - *Suppose that we dealing with several sensors deployed across multiple sites. We could put all of the data in one payload and parse it when it gets to its destination. Otherwise, we could use MQTT and use topics to subdivide the data, as shown below:*

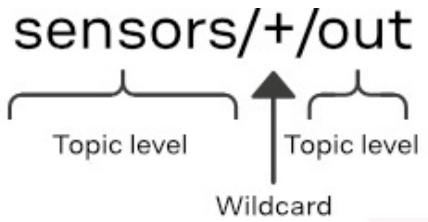
site1/position
site1/temp
site1/vibration
site2/position
site2/temp
site2/vibration

  - *If the transmitted data is divided by topic, "Things" can subscribe only to the topics they are interested in.*

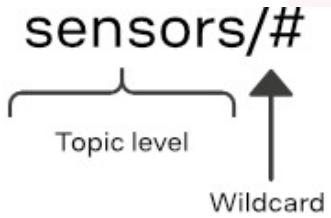
MQTT Topics (3)

- Subscription Wildcards:

- **Single-level** (+): It replaces one topic level
- *For example, that wildcard covers the following level:*
  - *sensors/soil/out*
  - *sensors/water/out*
  - *sensors/light/out*



- **Multi-level** (#): it replaces multiple topic levels
- *For example, that wildcard covers the following level:*
  - *sensors/soil/out*
  - *sensors/soil/in*
  - *sensors/temperature/out*



Every Ware Lab

QoS (Quality of Service)

- Quality of Service (**QoS**) in MQTT messaging is an **agreement** between **sender** and **receiver** on the guarantee of delivering a message.

- There are three levels of QoS:
  - 0 - at most once
  - 1 - at least once
  - 2 - exactly once

- Note that we have to consider the **two sides** of message delivery:
  - Message delivery **form the publishing** client to the **broker**.
  - Message delivery **from the broker** to the **subscribing** client.

QoS 0

- This is the **simplets**, **lowest-overhead** method of sending a message
- It provides the same guarantee as the underlying **TCP** protocol.
- It is suitable when you deal with a **reliable internet connection**

Every Ware Lab

QoS 1

- It guarantees that a message is **delivered at least one time** to the receiver.
- The **sender stores** the message until it gets a **PUBAK** packet from the receiver that acknowledges receipt of the message.
- Messages are also **queued on the broker** to ensure delivery to offline Clients.
- It is a good choice if your IoT application can **tolerate receiving** a **message more than once.**

Every Ware Lab

QoS 2

- It guarantees that a message is **delivered exactly once** to the receiver.
- Messages are also **queued on the broker** to ensure delivery to offline Clients.
- As it has a relatively **high cost** in terms of **data transfer** you should consider whether even a lower-cost QoS would be suitable

Every Ware Lab

Create a Broker

- **Mac OS**

Install homebrew from (https://brew.sh/index_it)
From the terminal:

brew install mosquito
brew services start mosquitto
… running …
brew services stop mosquitto

- **Windows**

Download and install the mosquitto broker from here:
https://mosquitto.org/download/

Launch the mosquitto.exe file from this path *C:\Program Files\Mosquitto*

- **Linux** (http://www.steves-internet-guide.com/install-mosquitto-linux/)

```
sudo apt-add-repository ppa:mosquitto-dev/mosquitto-ppa
sudo apt-get update
sudo apt-get install mosquitto
sudo apt-get install mosquitto-clients
sudo apt clean
sudo service mosquitto start
        ... running …
sudo service  mosquitto stop
```

Every Ware Lab

Connect a Client

- The connection is established by the client sending a **CONNECT** message, to which the broker responds with a **CONNACK** (connection acknowledged).

- Once the connection is established, the **broker keeps it open until the client sends a disconnect command** or the connection breaks.

Every Ware Lab

Persistent session

- If the **connection** between the client and broker is **interrupted**, the topics on which the client subscribed will be lost

- To avoid this problem, the client can request a **persistent session** when it connects to the broker (using ***cleanSession flag***)

- Persistent sessions **save all information** that is relevant for the client on the **broker**:
  - All the subscription topics of the client
  - All messages in a Qos1 or 2 that the client has not yet confirmed
  - All new QoS 1 or 2 messages that the client missed while offline
  - All QoS 2 messages received from the client that are not yet completely acknowledged

Last will and testament

- The Last Will and Testament provides a **way for clients to respond to ungraceful disconnects** in an appropriate way.
- Each **client** can **specify its last will message** in the CONNECT message
- The last will message is a **normal MQTT message** with a **topic**, retained message flag, QoS, and payload.
- If the **broker** detects that a client disconnected **ungracefully,** it sends the last-will message to all subscribed clients of the last-will message topic.

Java Package for MQTT

- Package org.eclipse.paho.client.mqttv3

https://www.eclipse.org/paho/files/javadoc/org/eclipse/paho/client/mqttv3/package-summary.html

- MqttClient Class

https://www.eclipse.org/paho/files/javadoc/org/eclipse/paho/client/mqttv3/MqttClient.html

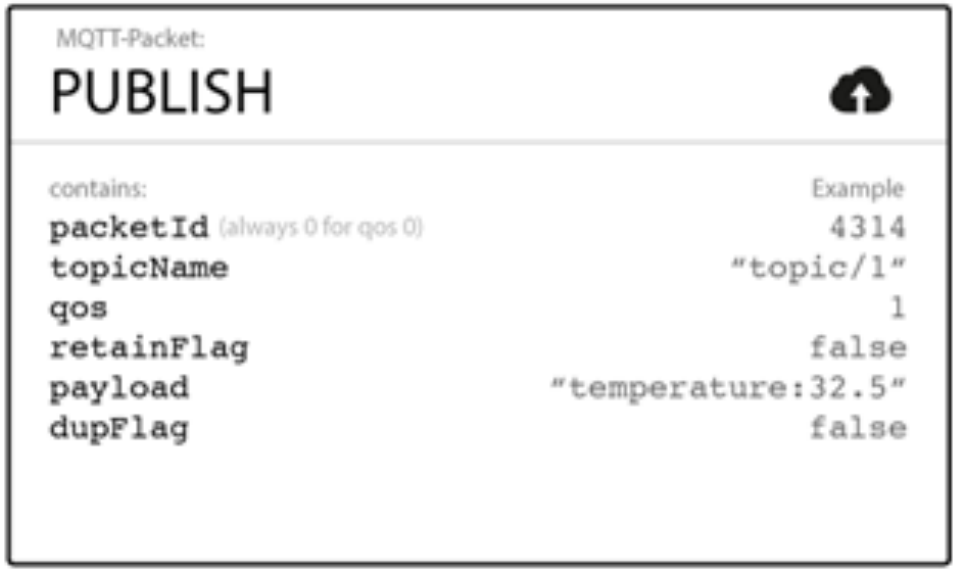
Create and Connect a MQTT Client

```
String broker = "tcp://localhost:1883"; // default MQTT broker address
String clientId = ”12345";


try {
    // Create an Mqtt client
    MqttClient mqttClient = new MqttClient(broker, clientId);
    MqttConnectOptions connOpts = new MqttConnectOptions();
    connOpts.setCleanSession(true);
    connOpts.setUserName(username); // optional
    connOpts.setPassword(password.toCharArray()); // optional
    connOpts.setWill(“this/is/a/topic”, "will message".getBytes(),1,false); // optional
    connOpts.setKeepAliveInterval(60); // optional

   // Connect the client to the broker (blocking)
   mqttClient.connect(connOpts);

} catch (MqttException me ) {

        // handle exceptions

}
```

Every Ware Lab

Publish a Message

- MQTT clients can publish messages as soon as it connects to a broker.
- Each message must contain a **topic.** The broker will use that topic to forward the message to the subscribed clients
- It is also essential to specify the **QoS** of the message
- Note that the **payload** must be specified in **binary format**



Every Ware Lab

Retained messages

- A retained message is a normal MQTT message with the **retained flag** set **true.**
- When a client subscribes to a topic that matches the topic of the retained message, it **receives** the retained message **immediately after the subscription**.
- The **broker** stores **only one** retained message **per topic.**
- To **delete** a retain message of a specific topic, a publisher sends a retained message having that topic and a zero-byte payload

Publish Example

```
// Create a Mqtt message
MqttMessage message = new MqttMessage(payload.getBytes());

// Set the QoS on the Message
message.setQos(qos);

// Set retain flag (optional)
message.setRetained(false);

// Publish the message (blocking)
mqttClient.publish(topic, message);
```

Every Ware Lab

MqttCallback()

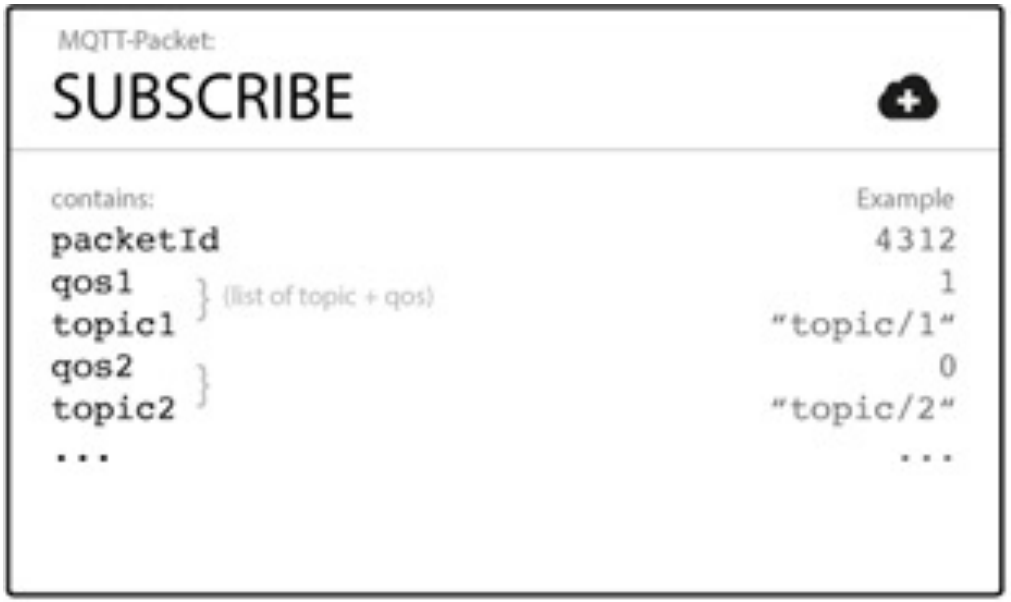
- It enables a MqttClient to work **asynchronously**
- Optional for publishing, **mandatory** for **subscribing**
- The client must set the MqttCallback() after connecting to the broker and before publishing a message or subscribing to a topic

```
mqttClient.setCallback(new MqttCallback() {

    public void messageArrived(String topic, MqttMessage message) {
        // handle incoming messages
    }

    public void connectionLost(Throwable cause) {
        // inform the client if the connection with the broker unexpectedly lost
    }

    public void deliveryComplete(IMqttDeliveryToken token) {
        // inform the client that a message will be delivered (or not) to the broker.
        }

});
```

Every Ware Lab

Subscribe

- The client must be already connected to the broker before subscribing to any topic
- The SUBSCRIBE request can include multiple topics
- The SUBSCRIBE request includes the **QoS setting** which can be used to **downgrade the QoS of the published** message.



Every Ware Lab

Subscribe Example

```
mqttClient.setCallback(new MqttCallback() {
 …
public void messageArrived(String topic, MqttMessage message) {
        String time = new Timestamp(System.currentTimeMillis()).toString();
        String receivedMessage = new String(message.getPayload());
        System.out.println(clientId +" Received a Message!"+
         "\n\tTime:    " + time +
         "\n\tTopic:   " + topic +
         "\n\tMessage: " + receivedMessage +
         "\n\tQoS:     " + message.getQos() + "\n");
    }

});
```

- Subscribe the mqttClient to a single topic with a specific QoS

  mqttClient.subscribe(*"topic/a", qos*);

- Subscribe the mqttClient to multiple topics, each having a specific QoS

  mqttClient.subscribe(*["topic/a", topic/b"] , [qos_a, qos_b]*);

Every Ware Lab

MqttAsyncClient

- The *MqttClient* class that we presented in this lesson is the synchronous variant of the the *MqttAsyncClient* class

- *MqttClient* Implementation: https://github.com/eclipse/paho.mqtt.java/blob/master/org.eclipse.paho.client.mqttv3/src/main/java/org/eclipse/paho/client/mqttv3/MqttClient.java

- In particular, the MqttClient implementation “wraps” some asynchronous methods (e.g., connect(), publish()) of the MqttAsyncClient class, and makes them synchronous.

```
public void connect(MqttConnectOptions options) throws MqttSecurityException, MqttException {
        aClient.connect(options, null, null).waitForCompletion(getTimeToWait());
}
```

Every Ware Lab

Exercise

- Create *3* processes that simulate a temperature sensor that every 5 seconds publishes a random temperature value (between 18 and 22 degrees) to the topic *"home/sensors/temp"*

- Create a process that subscribes to the topic *"home/sensors/temp"* and computes the average of the last 5 sensors measurements. If that average temperature exceeds 20 degrees, it sends a message to the topic *"home/controllers/temp"* that signals to turn off the heaters. Otherwise, it signals to turn them on.

- Create one process that simulates a heater which subscribes to the topic *"home/controllers/temp"* and that prints in the console when it turns on or off.

Every Ware Lab

References

- Code Examples and setup:

https://ewserver.di.unimi.it/gitlab/riccardopresotto/setup_rest_mqtt

Every Ware Lab