**IoT’s:**

**Protocols:**

**1. FTP (File Transfer Protocol)**

**2. one M2M ()**

**3. LW M2M (Light Weight Machine to Machine)**

**4. OMA-DM (Open Mobile Alliance-Device Management)**

**5. TR06 ()**

**6. MQTT (Message Queuing Telemetry Transport)**

**===================================================================**

**HTTP, MQTT, CoAP, WS**

**CoAP (**Constrained Application Protocol**):**

CoAP, Constrained Application Protocol, is a RESTful application protocol running over UDP that is used for resource-constrained, low-power devices in lossy networks, especially optimized for deployments with a high number of end devices within the network.

===================================================================

**Local Connectivity:**

**1. NFC (Near-Field Communication)**

Near Field Communication (NFC) is a set of short-range wireless technologies, typically requiring a distance of 4cm or less to initiate a connection.

**2. BLE (Bluetooth Low Energy)**

Bluetooth Low Energy is a wireless, low-power personal area network that operates in the 2.4 GHz ISM band. Its goal is to connect devices over a relatively short range.

**What is Server?**

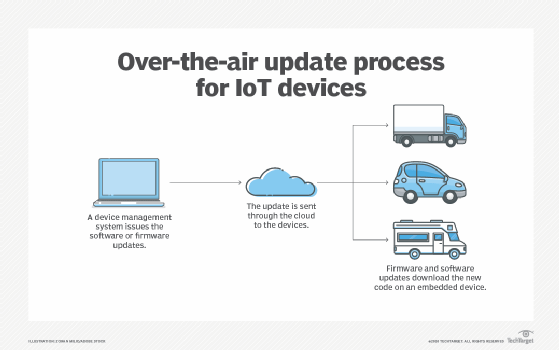
A **server** is a software or hardware device that accepts and responds to requests made over a network.

The device that makes the request, and receives a response from the server, is called a [client](https://www.computerhope.com/jargon/c/client.htm).

On the Internet, the term "server" commonly refers to the computer system that receives requests for a web files and sends those files to the client.

**Over The Air (OTA):**

workflow for IoT software updates



**Firmware Image:**

The firmware image, or simply the "image", is a binary that may contain the complete software of a device or a subset of it. Thefirmware image may consist of multiple images if the device contains more than one microcontroller. Often, it is also a compressed archive that contains code, configuration data, and even the entire file system. The image may consist of a differential update for performance reasons.

The terms "firmware image", "firmware", and "image" are used in this document and are interchangeable. We use the term "application firmware image" to differentiate it from a firmware

image that contains the bootloader. An application firmware image, as the name indicates, contains the application program often including all the necessary code to run it (such as protocol stacks and an embedded operating system (OS)).

**Remote Firmware Updates for IoT Devices**

As more and more resource-constrained IoT devices are being deployed, firmware updates over the air (FOTA) are becoming increasingly important. As physical access is often expensive, or even impossible to realize when operating large deployments in remote, hard-to-reach locations, FOTA provides a way to fix bugs, patch security vulnerabilities, or add functionality throughout the lifespan of a connected device.

Lightweight M2M (LwM2M) defines the process for remote firmware updates in detail. When following the standardized guidelines provided by the protocol, updating the firmware of resource-constrained devices remotely can be accomplished effortlessly, regardless of the device or platform used.

## **The LwM2M way of updating firmware**

Every platform or device vendor can develop a custom FOTA mechanism, also known as **FUOTA** (Firmware Update OTA) or **DFU** (Device Firmware Upgrade). As there is no globally unified method to update device firmware, many different implementations can be found on the market.

Embedded developers can simply get one of the available libraries from their specific hardware vendor or cloud service provider to manage the update process. However, these implementations are specific to the vendor or platform and cannot be easily ported to other systems. Adopting a new platform usually means redesigning the FOTA process from scratch.

The problem of vendor compatibility can be resolved by adopting the LwM2M standard, which is hardware agnostic and can be implemented on any hardware platform. The standard provides clear guidelines on how devices should report their data, how remote configuration can be performed using server commands, and how firmware updates can be executed, regardless of the device manufacturer, firmware version, or hardware platform used.

**Firmware update process**

Generally speaking, the LwM2M firmware update process consists of four steps:

1. The IoT device (referred to as the LwM2M Client) is triggered to **initiate the firmware update process.**
2. The LwM2M Client **downloads the firmware** and reports to the LwM2M Server when the download is finished.
3. The LwM2M Client **performs the firmware update** after validating the integrity and authenticity of the new firmware which is done through a process called **secure boot**.
4. The Client attempts to **run the new firmware** and reports the status to the Server. If succeeded, the device starts running the new firmware, if an error is encountered the device performs a rollback to the earlier firmware version.

This procedure is defined in detail in the LwM2M specifications (see [LwM2M specifications](https://www.openmobilealliance.org/release/LightweightM2M/V1_1_1-20190617-A/HTML-Version/OMA-TS-LightweightM2M_Core-V1_1_1-20190617-A.html#13-6-1-0-E61-Firmware-Update-State-Machine)). To add firmware update capabilities to a device, the [Firmware Update Object /5](https://devtoolkit.openmobilealliance.org/OEditor/Legal?back=LWMOView) needs to be implemented containing all essential functionalities for conducting the update and reporting the status. Several LwM2M Clients natively support this Firmware Update Object, including AVSystem’s [Anjay](https://www.avsystem.com/products/anjay/).

**Downloading new firmware**

Timing is critical when it comes to firmware updates. If the radio signal quality is poor, transferring data can take a lot of time due to low throughput caused by lost data packets that need to get retransmitted. Since the device’s radio module must be in the connected state for a long time, downloading the file in such circumstances results in a faster battery drain.

The LwM2M standard defines two methods to perform a firmware update, allowing either the Server or the Client to decide on the best moment to initiate the firmware download process.

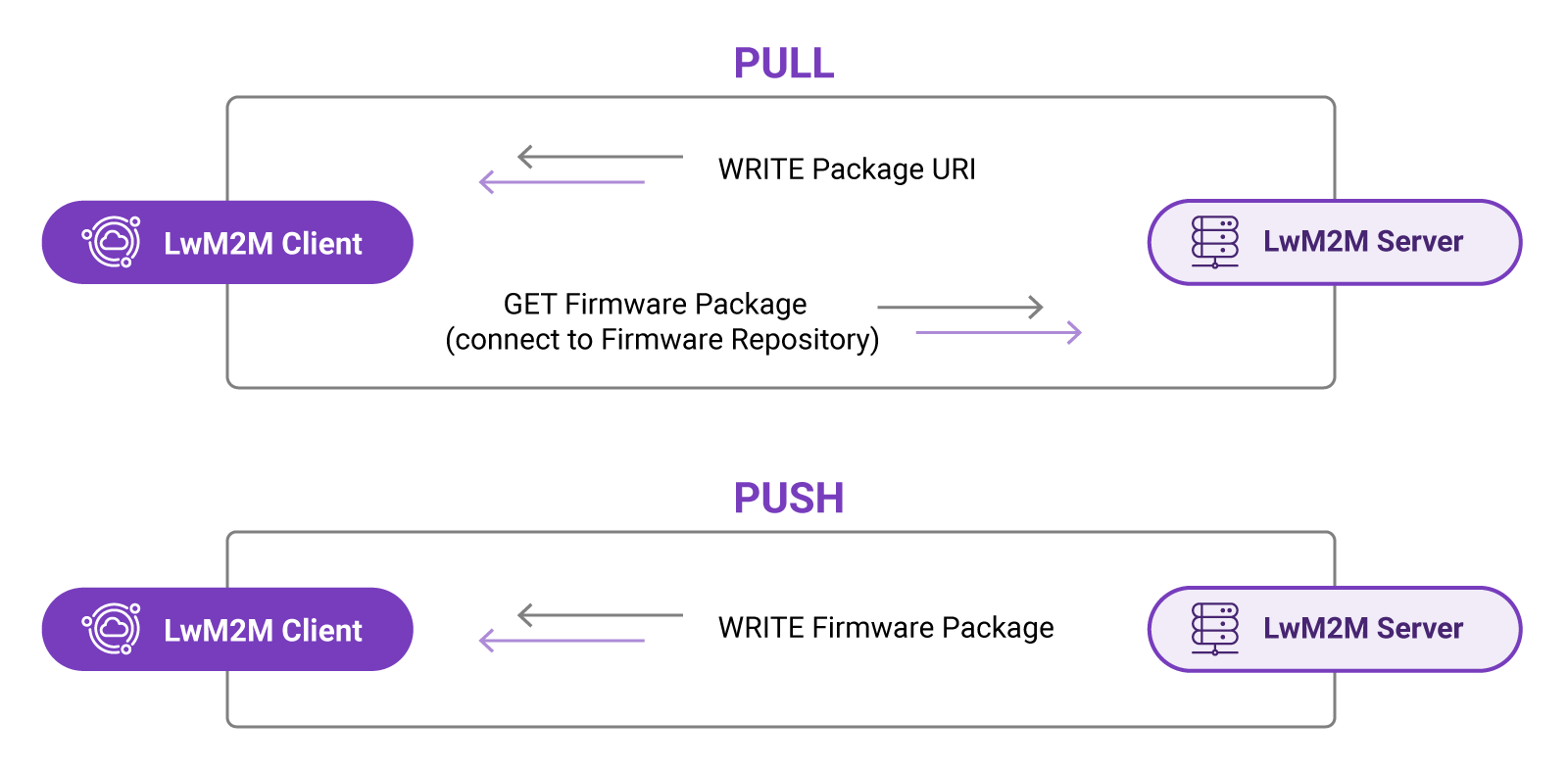
**The two methods are PULL and PUSH.**

1. **PULL method:** A device receives the location of the file that is to be downloaded and pulls the file from it.
2. **PUSH method:** A LwM2M Server pushes the firmware file to the device.

**PULL method**  
In the PULL scenario, the LwM2M Server provides the device with the address of the server containing the firmware file known as **the package URI**. The device subsequently downloads the firmware from the so-called firmware repository at the earliest available opportunity.

**PUSH method**  
In the PUSH scenario, the LwM2M Server determines the moment to initiate the firmware download, ideally based on the connectivity conditions. The device must have access to information such as Radio Signal Strength and Link Quality to support such server-side decision-making these resources are available in the **LwM2M Connectivity Monitoring Object** /4.

### *Firmware Delivery Methods*



**Firmware Update Object**

The FOTA process is defined in the **Firmware Update Object** /5. This Object defines the update process using **four states**(defined by Resource/5/\*/3), as well as **multiple update results(**defined by Resource/5/\*/5) representing the most common outcomes of the firmware update process.

**OTA update modes**

As for selecting the right OTA architecture, the one size fits all OTA update mode doesn’t fit anybody. Or does it? It all depends. We’re talking about IIoT, so the answer would be “it does”. For IoT devices in general, it would depend on the use case. But for IIoT devices there is a definitive advantage on using the push-pull mode. For completeness I discuss the other update modes as well, but all remarks mentioned are IIoT in mind and not IoT.

### Push

In this mode, the cloud (or local) update server pushes a new firmware version to each device.

The biggest advantage of this mode is that the updates are immediate. This means that the device is in a state where the update can be applied.

The biggest disadvantage is that this is a very bandwidth consuming mode. Each update needs to be send to each individual device, making it only useful for updating small “fleets” of devices.

### Pull

In a pull mode scenario, the device itself checks on a regular basis whether there is an upgrade available and if so, downloads the upgrade.

The advantage of this scenario is that it consumes considerable less bandwidth than the push mode.

The biggest disadvantage is then again the pulling, if the pull interval is long, this might be an issue for critical updates. Then again, if a device is not in a safe state to update it might take long as well before the update is applied.

The second disadvantage is that a device does not need to register for the updates. One might consider that this is an advantage because it is much more simple to develop, but from a device management and security perspective, this is not desired.

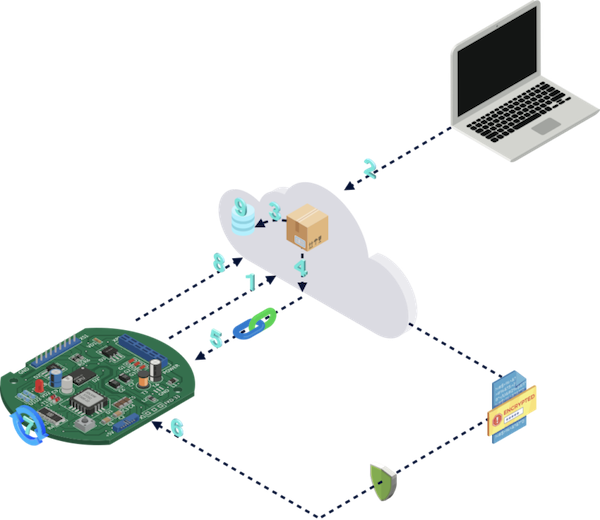
### Push-Pull

In an industrial environment, the best of both worlds, the push-pull OTA mode is the preferred mode.

So if this is the preferred mode why would one bother implementing others? Implementing a push-pull mode OTA requires a much more complex and costly development, so if you are just getting started and only a handful of devices need to be managed and upgraded, a push OTA update might make more sense since the time to market will be faster and the required development budget will be lower.

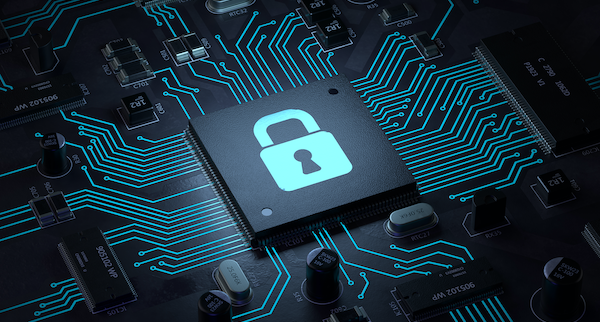
After all, the OTA process should be able to update itself and switch to a push-pull mode when it receives that update as your number of devices grow (and so will your development budget, making the push-pull OTA mode accountable to your management).

Since this is by far the most important IIoT OTA update mode, the process is described in detail below.



1. The device connects to the update cloud and registers for updates. The update server is typically a message broker (e.g. an MQTT broker).
2. A new firmware, configuration, application, … is committed on the upgrade server (cloud). This can either be for all devices, a fleet of devices, or a specific device.
3. The new firmware availability for the devices gets stored in a database for audit purposes.
4. A temporary link to the new firmware gets created for each device (only available with the devices certificate) and the link of the update gets pushed on the update broker (where the device registered on).
5. The device receives an event that there is a new firmware version available, together with the temporary link to that firmware version.
6. The device downloads the new firmware over a secure channel. It checks the downloaded update for consistency, decrypts end decompresses it and puts it on the landing zone for new firmware. When this is done, the firmware update server gets informed by the device that the new firmware has been downloaded, so that the temporary link may be removed.
7. When the device is in a safe state it can perform the update. Safe means its context says it is ok to perform the update at that given point in time
8. The device notifies the update server that the update has been executed.
9. The update server registers the execution of the update in the database for full traceability.

**OTA security**



While a connected IIoT devices OTA feature introduces many advantages, ranging from bug fixes to introducing new functionality, it also poses security concerns.

Nobody wants to receive an update from another device, let alone a malicious update. So it is very important to have security built-in from the start in IIoT OTA update mechanisms.

Here are several guidelines that can be applied for security.

### Secure communication channel

Always use a secure encrypted communication channel between the IIoT device and the cloud to update the server and firmware downloads.

When using a gateway, apply the same security standards for the local communication between the gateway and the edge devices.

Managing certificates might look like a painful administrative task. Still we encourage to use an individual certificate for each device and not to use the same certificate for multiple devices.

### Temporary links

The link to the new firmware should be relative to the device itself so that anyone else (or something else) with another certificate can not download the new firmware.

Moreover, restrict the link to the new firmware in time. And after download of the update, remove the link.

In case the firmware isn’t downloaded within a preconfigured timeframe, delete the link. In such an event a new update link must be generated and pushed to the device.

### Encryption

It’s not sufficient that all communication uses an encrypted communication channel like https.

Also each update file and each link should be encrypted as well. This prevents the ability to read and decrypt the updates if the communication channel its security would be compromised.

### Code provenance & Signed updates

Each update must be cryptographic code signed so that the device receiving (downloading) the update only accepts the update if it can verify:

* the origin of the update
* the destination of the update is indeed the device itself
* the update has not been altered during transit

**OTA management**



If you have 1 device or 10 devices, managing these updates still is a pretty easy straight forward job. But imagine you need to do this for 4000 devices or more.

Indeed, that’s right: how do we manage this? That’s the question a lot of people ask and get confronted with.

In case of a limited amount of devices, sending an update to all devices still might look like a good idea. For the 4000 devices everyone understands this might not be the desired mechanism, keeping Werner Vogels statement in mind: design for failure!

### Fleet & campaign management

Even if your update succeeded and you have tested it all well, there still might go something wrong with your new version in production.

That is why you do not want to update the 4000 devices at the same time. Instead you want to have multiple campaigns of updates for the same firmware release.

You start with a group of devices and if all works well for these devices, you gradually update all other devices in groups.

Since you want to use campaigns and also want to do it customer by customer or production section by section, … you also want a fleet management in place before you start creating update campaigns.

### Scaleability

Make sure your OTA solution is capable of handling the updates of a few devices to hundreds, thousands or even hundreds of thousands of devices.

You might start small, but if your business model works or your management sees the benefit of having updates rolled out at ease, you end up quickly with a large number of devices to update.

And it would be a shame shining in the possibility of remotely updating devices, but failing in upscaling your solution.

### OTA levels (config, firmware, app)

A firmware update might seem like a big thing but our remote update might as well be a configuration change of a device. Remote updates are not only applicable to firmware updates. They also apply to new configurations as well.

### Code compatibility & verification

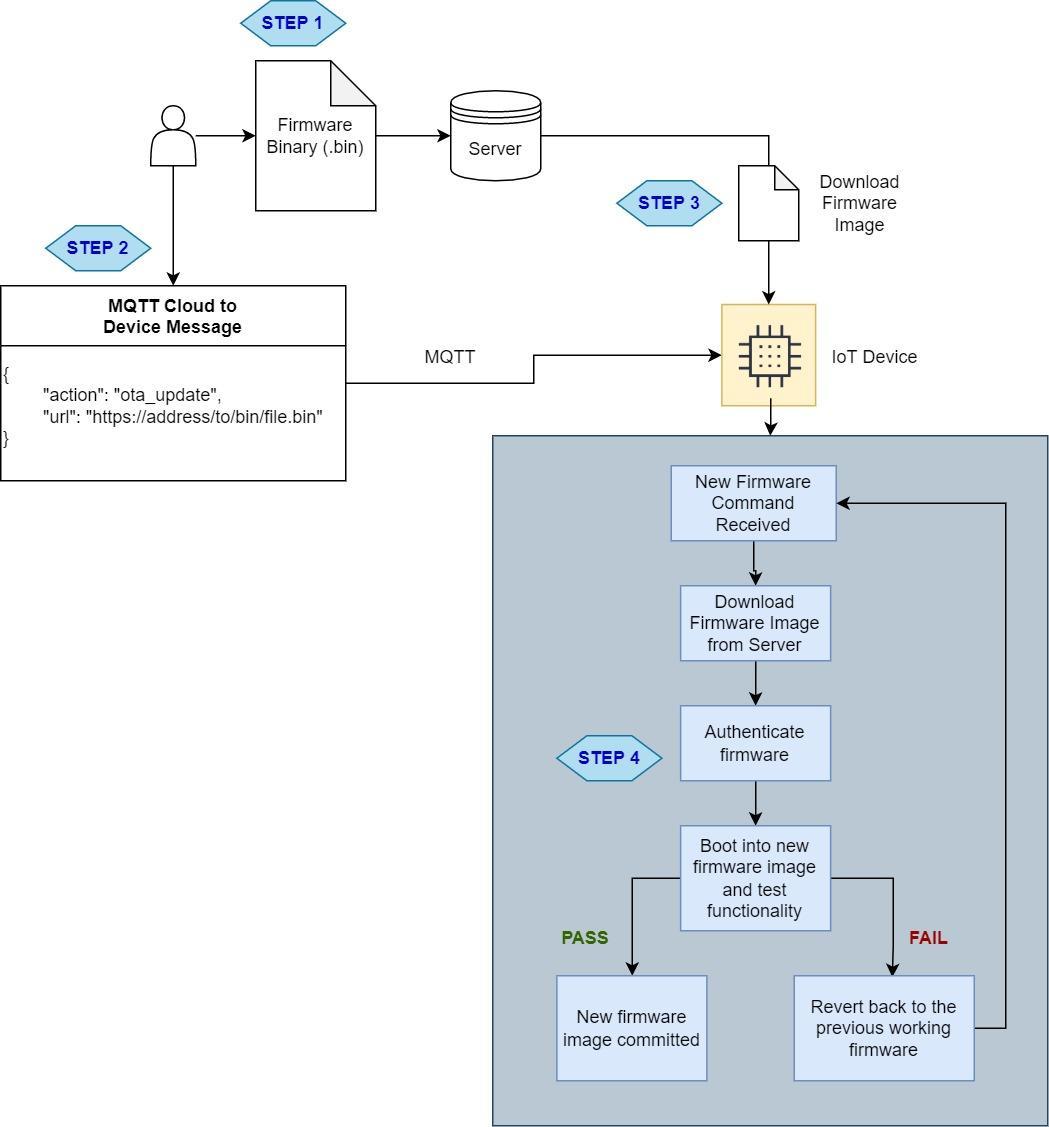
Let’s assume your fleet of devices is composed out of multiple MCU architectures. When you distribute firmware updates, it is advisable that your update contains the architecture it is designed for.

So if you receive a firmware designed for an esp32 and the device itself is an ARM32, the device will not update and report the error to the update server.

### Emergency updates

In the case of a very high variability risk, you want to roll out an emergency update from the moment it is available. In an emergency situation it might be more important to patch the variability then to have the device check its state context.

Generally, the firmware upgrade process looks something like this:



**Step 1:**The binary files are uploaded to a server and a URL is obtained to point to the binary file.

**Step 2:**A notification is sent to the IoT device that new firmware is available for download, and the URL for the firmware is provided.

**Step 3:**IoT device will download the firmware file using HTTP. The firmware file should be saved in a different partition (more on this shortly).

**Step 4:**IoT device will check the authenticity of the binary files and will start to load in the newly downloaded firmware.

A fallback mechanism should be in place after steps 3 and 4. In case the firmware file is corrupted during download, the IoT device should not load in the new firmware or free the used resources.

# MQTT integration with spring boot

# MQTT is a publish/subscribe protocol that allows edge-of-network devices to publish to a broker. Clients connect to this broker, which then mediates communication between the two devices. … When another client publishes a message on a subscribed topic, the broker forwards the message to any client that has subscribed.

# 

# One of Most Common Example of MQTT

**The Most Popular MQTT Broker are**

1. Mosqitto
2. [VerneMq](https://vernemq.com/)
3. RabbitMq
4. EMQ
5. [hivemq](https://www.hivemq.com/)

And MQTT Broker run on **1883** as its default port.

# MQTT: The Standard for IoT Messaging

MQTT is an OASIS standard messaging protocol for the Internet of Things (IoT). It is designed as an extremely lightweight publish/subscribe messaging transport that is ideal for connecting remote devices with a small code footprint and minimal network bandwidth. MQTT today is used in a wide variety of industries, such as automotive, manufacturing, telecommunications, oil and gas, etc

# Spring Boot With MQTT

**Step1** . Setup a Maven Project With eclipse poho dependency into spring boot pom.xml file in project.

<dependency>  
 <groupId>org.eclipse.paho</groupId>  
 <artifactId>org.eclipse.paho.client.mqttv3</artifactId>  
 <version>1.2.0</version>  
 </dependency>

Step 2. Create Mqtt configuration class where we have broker credentials broker address etc.

**public abstract class** MqttConfig {  
   
 **protected final** String **broker** = **"your broker address or IP "**;  
 **protected final int qos** = 1;  
 **protected** Boolean **hasSSL** = **false**; /\*By default SSL is disabled \*/ **protected** Integer **port** = 1883; /\* Default port \*/ **protected final** String **userName** = **"your username"**;  
 **protected final** String **password** = **"Password"**;  
 **protected final** String **TCP** = **"tcp://"**;  
 **protected final** String **SSL** = **"ssl://"**;  
  
 /\*\* \* Custom Configuration \*  \* **@param broker** \* **@param port** \* **@param ssl** \* **@param withUserNamePass** \*/ **protected abstract void** config(String broker, Integer port, Boolean ssl, Boolean withUserNamePass);  
  
 /\*\* \* Default Configuration \*/ **protected abstract void** config();  
   
  
}

Step 3. we can create a class when we can fetch data after subscribe to mqtt topic on which data is coming. And to annotate with @component

@Component  
**public class** MqttSubscriberImpl **extends** MqttConfig **implements** MqttCallback{  
  
 **private static final** String **fota\_fetch\_record** = **"fota\_fetch\_record"**;  
 **private** String **brokerUrl** = **null**;  
 **final private** String **colon** = **":"**;  
 **final private** String **clientId** = UUID.randomUUID().toString();  
  
 **private** MqttClient **mqttClient** = **null**;  
 **private** MqttConnectOptions **connectionOptions** = **null**;  
 **private** MemoryPersistence **persistence** = **null**;  
  
 **private static final** Logger **logger** = LoggerFactory.getLogger(MqttSubscriberImpl.**class**);  
  
 **public** MqttSubscriberImpl() {  
 **logger**.info(**"I am MqttSub impl"**);  
 **this**.config();  
 }  
  
 @Override  
 **public void** connectionLost(Throwable cause) {  
 **logger**.info(**"Connection Lost"** + cause);  
 **this**.config();  
 }  
  
 @Override  
 **protected void** config(String broker, Integer port, Boolean ssl, Boolean withUserNamePass) {  
 **logger**.info(**"Inside Parameter Config"**);  
 String protocal = **this**.**TCP**;  
  
 **this**.**brokerUrl** = protocal + **this**.**broker** + **colon** + port;  
 **this**.**persistence** = **new** MemoryPersistence();  
 **this**.**connectionOptions** = **new** MqttConnectOptions();  
  
 **try** {  
 **this**.**mqttClient** = **new** MqttClient(**brokerUrl**, **clientId**, **persistence**);  
 **this**.**connectionOptions**.setCleanSession(**true**);  
 **this**.**connectionOptions**.setPassword(**this**.**password**.toCharArray());  
 **this**.**connectionOptions**.setUserName(**this**.**userName**);  
 **this**.**mqttClient**.connect(**this**.**connectionOptions**);  
 **this**.**mqttClient**.setCallback(**this**);  
 } **catch** (MqttException me) {  
 **throw new** com.bms.exceptions.MqttException(**"Not Connected"**);  
 }  
 }  
  
 @Override  
 **protected void** config() {  
 **logger**.info(**"Inside Config with parameter"**);  
 **this**.**brokerUrl** = **this**.**TCP** + **this**.**broker** + **colon** + **this**.**port**;  
 **this**.**persistence** = **new** MemoryPersistence();  
 **this**.**connectionOptions** = **new** MqttConnectOptions();  
 **try** {  
 **this**.**mqttClient** = **new** MqttClient(**brokerUrl**, **clientId**, **persistence**);  
 **this**.**connectionOptions**.setCleanSession(**true**);  
 **this**.**connectionOptions**.setPassword(**this**.**password**.toCharArray());  
 **this**.**connectionOptions**.setUserName(**this**.**userName**);  
 **this**.**mqttClient**.connect(**this**.**connectionOptions**);  
 **this**.**mqttClient**.setCallback(**this**);  
 } **catch** (MqttException me) {  
 **throw new** com.bms.exceptions.MqttException(**"Not Connected"**);  
 }  
 }  
  
 @Override  
 **public void** subscribeMessage(String topic) {  
 **try** {  
  
 **this**.**mqttClient**.subscribe(topic, **this**.**qos**);  
 } **catch** (MqttException me) {  
 System.**out**.println(**"Not able to Read Topic "**+ topic);  
 // me.printStackTrace(); }  
 }  
  
 @Override  
 **public void** disconnect() {  
 **try** {  
 **this**.**mqttClient**.disconnect();  
 } **catch** (MqttException me) {  
 **logger**.error(**"ERROR"**, me);  
 }  
 }  
  
  
 @Override  
 **public void** messageArrived(String mqttTopic, MqttMessage mqttMessage) **throws** Exception {  
 String time = **new** Timestamp(System.currentTimeMillis()).toString();  
 System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"); System.out.println("Message Arrived at Time: " + time + " Topic: " + mqttTopic + " Message: " + new String(mqttMessage.getPayload()));// System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");   
 }  
  
 @Override  
 **public void** deliveryComplete(IMqttDeliveryToken token) {  
  
 }  
}

Step 4. To **subscriber** we have create Listener which sync in subscribe when a data set publish by any publisher or any IOT device

@Component  
**public class** MqttMessageListener **implements** Runnable {  
 @Autowired  
 MqttSubscriberApi **subscriber**;  
  
 @Override  
 **public void** run() {  
 **while** (**true**) {  
 **subscriber**.subscribeMessage(**"your mqtt topic name"**);  
 }  
  
 }

Step 5. We also need to implement Runnable task to run this Task when spring boot application will bootstrap

@SpringBootApplication  
**public class** Application **extends** SpringBootServletInitializer {  
  
  
 @Autowired  
 Runnable **MessageListener**;  
  
 @Override  
 **protected** SpringApplicationBuilder configure(SpringApplicationBuilder application) {  
 **return** application.sources(Application.**class**);  
 }  
  
 **public static void** main(String[] args) {  
 SpringApplication.run(Application.**class**, args);  
 }  
  
 @Bean  
 **public** CommandLineRunner schedulingRunner(TaskExecutor executor) {  
 **return new** CommandLineRunner() {  
 **public void** run(String... args) **throws** Exception {  
 executor.execute(**MessageListener**);  
 }  
 };  
 }

**Finally we done for subscribe part now we check how we can publish data over the Mqtt Protocol to send broker when other device can subscribe**

@Component  
**public class** MqttPublisherImpl **extends** MqttConfig **implements** MqttCallback{  
  
 **private static final** Logger **logger** = LoggerFactory.getLogger(MqttPublisherImpl.**class**);  
  
  **private** MqttPublisherImpl() {  
 **this**.config();  
 } **private** MqttPublisherImpl(String broker, Integer port, Boolean ssl, Boolean withUserNamePass) {  
 **this**.config(broker, port, ssl, withUserNamePass);  
 }  
 **public static** MqttPublisherImpl getInstance() {  
 **return new** MqttPublisherImpl();  
 }  
  
  **public static** MqttPublisherImpl getInstance(String broker, Integer port, Boolean ssl, Boolean withUserNamePass) {  
 **return new** MqttPublisherImpl(broker, port, ssl, withUserNamePass);  
 }  
  
 @Override  
 **public** void publishMessage(String topic, String message) {  
  
 **try** {  
 MqttMessage mqttmessage = **new** MqttMessage(message.getBytes());  
 mqttmessage.setQos(**this**.**qos**);  
 mqttmessage.setRetained(**false**);  
 **this**.**mqttClient**.publish(topic, mqttmessage);  
 } **catch** (MqttException me) {  
 **logger**.error(**"ERROR"**, me);  
 }  
 **return null**;  
 }  
 @Override  
 **public void** disconnect() {  
 **try** {  
 **this**.**mqttClient**.disconnect();  
 } **catch** (MqttException me) {  
 **logger**.error(**"ERROR"**, me);  
 }  
 }  
 @Override  
 **public void** connectionLost(Throwable arg0) {  
 **logger**.info(**"Connection Lost"**);  
  
 }  
 @Override  
 **public void** deliveryComplete(IMqttDeliveryToken arg0) {  
 **logger**.info(**"delivery completed"**);  
  
 }  
  
  @Override  
 **public void** messageArrived(String arg0, MqttMessage arg1) **throws** Exception {  
 // Leave it blank for Publisher }@Override  
 **protected void** config(String broker, Integer port, Boolean ssl, Boolean withUserNamePass) {  
 // Like we did in MqttSubscribe  
 }  
  
 @Override  
 **protected void** config() {  
 // Like we did in MqttSubscribe   
 }  
   
  
 }

There are Many MQTT client are available as **[MQTT.fx](https://mqttfx.jensd.de/" \t "_blank)** , **[mqttBox](http://workswithweb.com/mqttbox.html" \t "_blank)**

which we can you use to testing Mqtt Data transfer like publish and subscribe