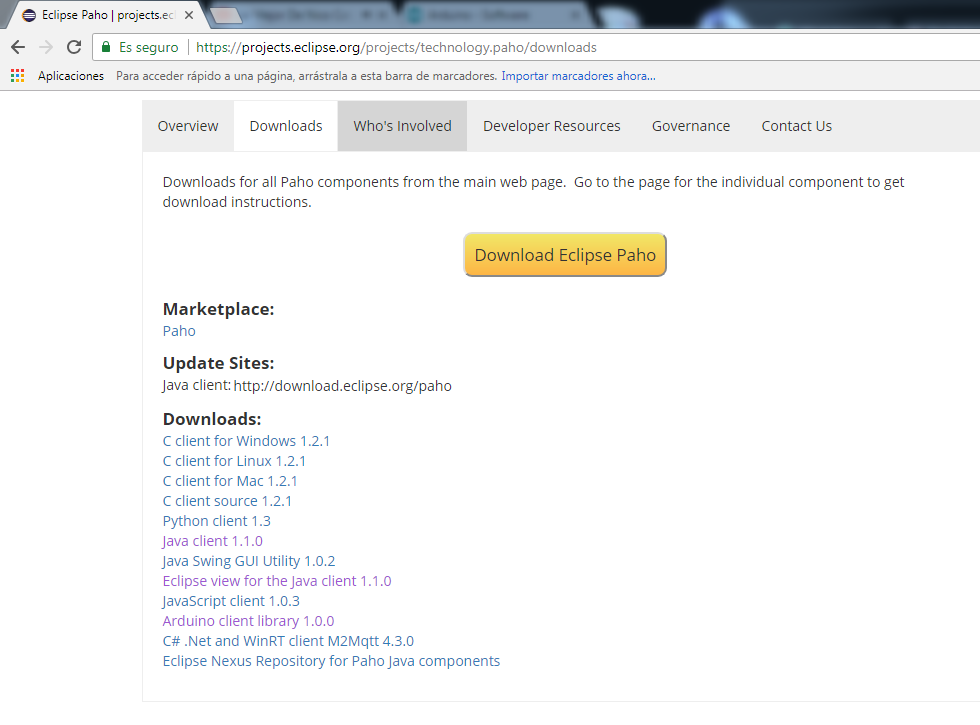
Descargamos la implementación de Paho para Arduino:

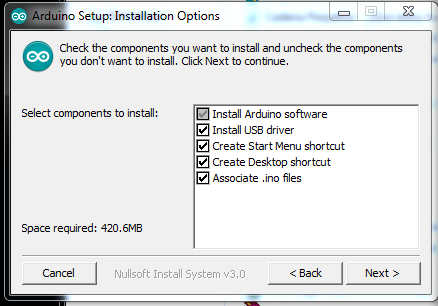


Y el IDE para Arduino desde:

<https://www.arduino.cc/en/Main/Software>



Al instalar el IDE:



Rest Service (Web)

<https://spring.io/guides/gs/rest-service/>

<https://docs.spring.io/spring-boot/docs/current/reference/html/getting-started-first-application.html>

Descargar:

Maven: <https://maven.apache.org/download.cgi>

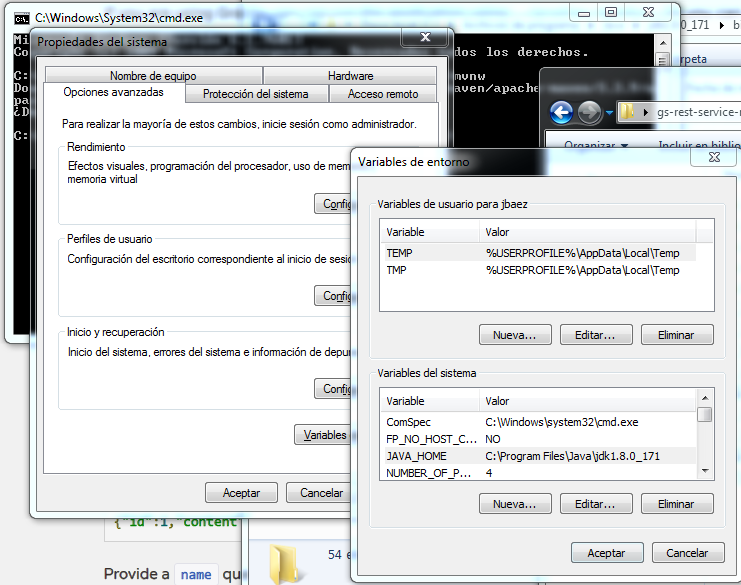
Java JDK: <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

Eclipse IDE:

<https://www.eclipse.org/downloads/>

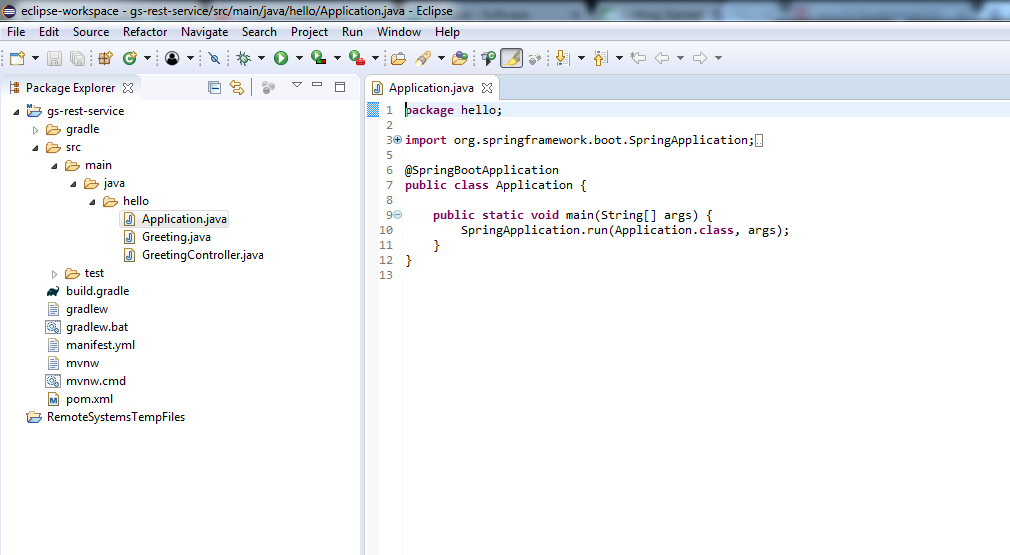
\*Al instalar Eclipse IDE, nos debemos asegurar que instalemos la adaptación para “Java EE Developers” ya que tiene soporte para Servidor de aplicaciones.

Agregar la variable de entorno JAVA\_HOME para desarrollar en Java:

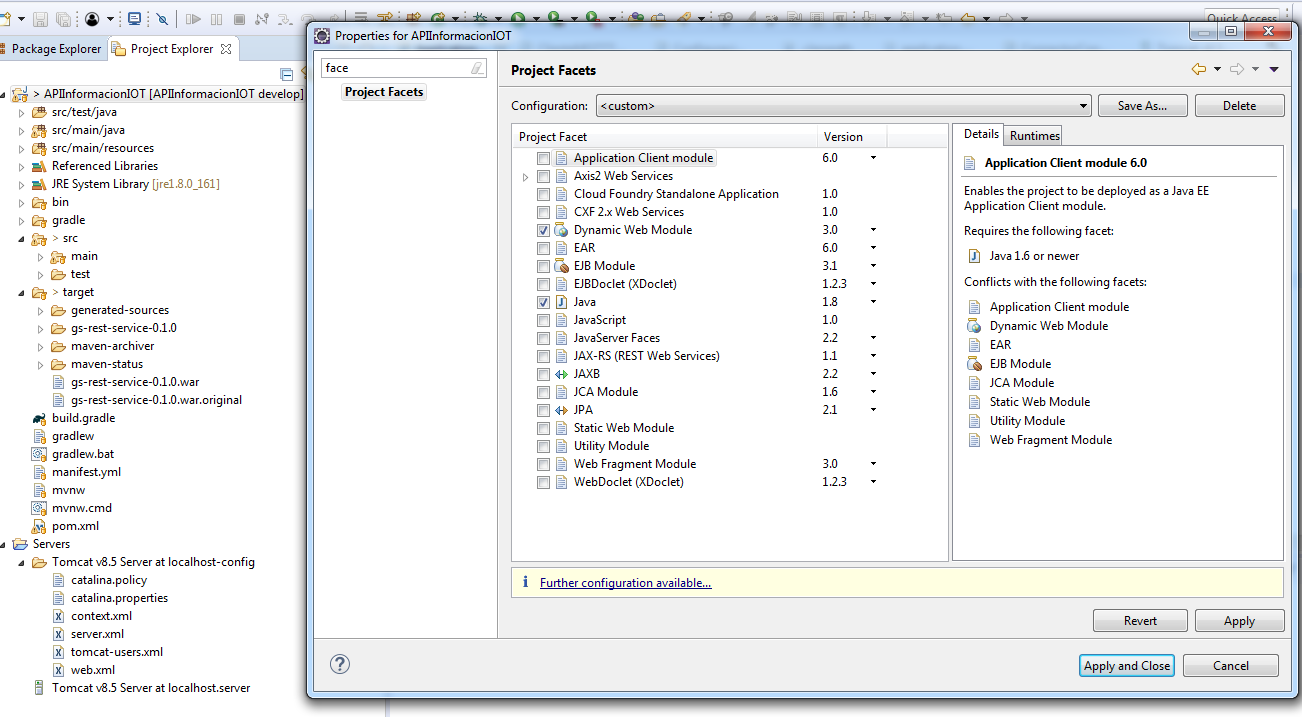


Importamos el ejemplo de API Rest:

Import → Maven → Existing Maven Projects (ya que es un proyecto organizado con Maven)

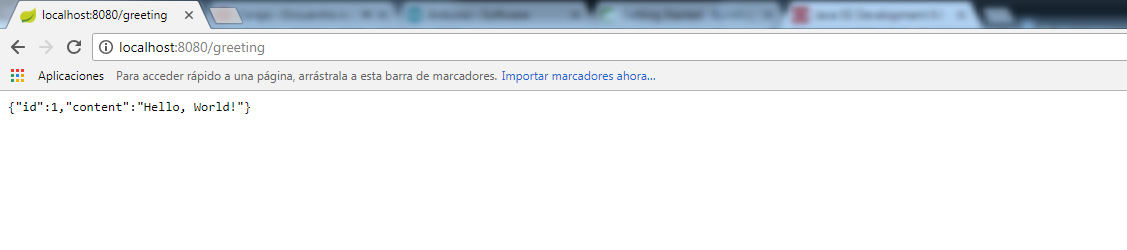


Para deployar una APP en el Tomcat integrado del Eclipse, se debe configurar el proyecto para el propósito o facet Web:



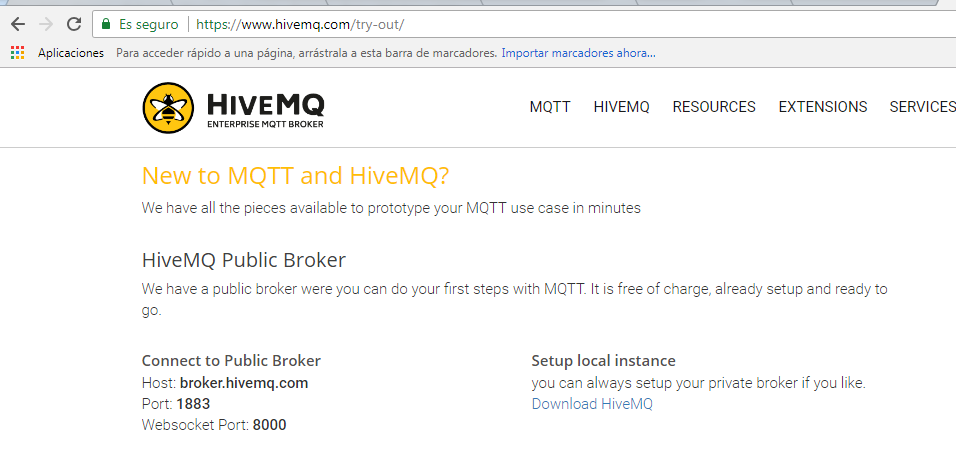
Seguimos el tutorial y levantamos nuestra API rest Prediseñada:

mvnw spring-boot:run (levanta un apache )



Comunicación con HiveMQ:

<https://www.hivemq.com/blog/restful-http-apis-with-hivemq>



WebSocket:

<https://stackoverflow.com/questions/26452903/javax-websocket-client-simple-example>

Descargamos un servidor de aplicaciones Java:

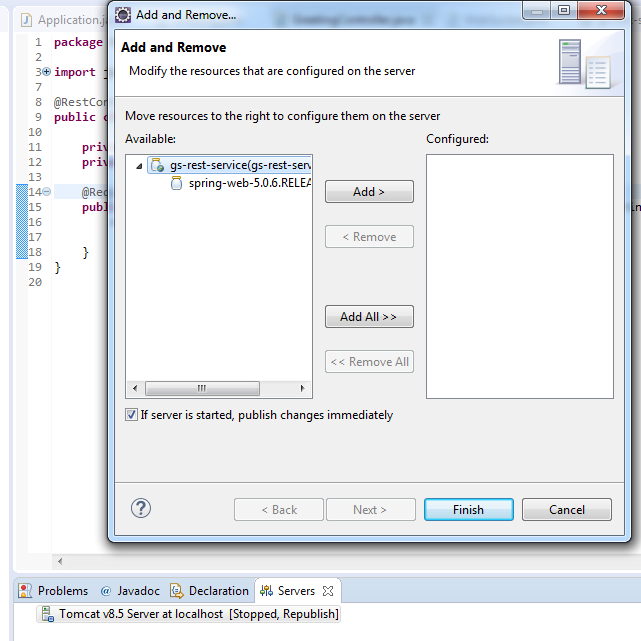
En este caso Tomcat:

<http://tomcat.apache.org/download-80.cgi>

¿Cómo generar un WAR para deployar en Tomcat con Spring Boot?

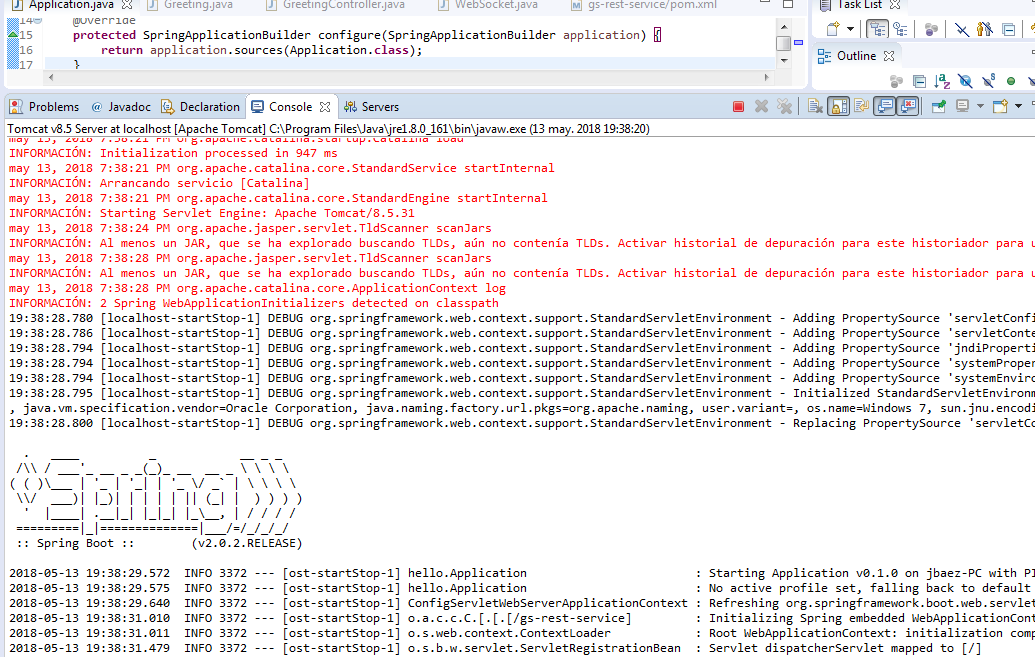
<https://www.mkyong.com/spring-boot/spring-boot-deploy-war-file-to-tomcat/>

Compilar el proyecto con Maven y agregar el proyecto al Tomcat:

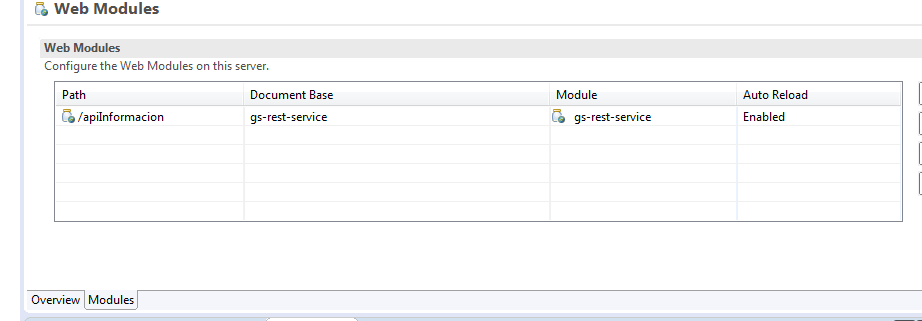


\*No estará disponible la opción hasta que se compile con mvn install.

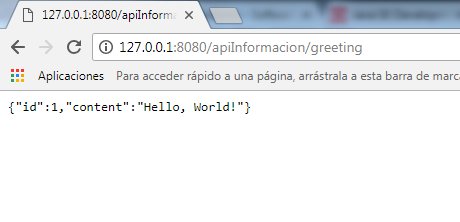
Iniciamos el Servidor de aplicaciones:



Configuramos el path de escucha para el Tomcat:



Hacemos request HTTP al Tomcat para ver si funciona:



Ahora implementar en la clase Application

<https://www.hivemq.com/blog/mqtt-client-library-encyclopedia-eclipse-paho-java>

Crear hilo de conexión con MQTT:

<http://tutorials.jenkov.com/java-concurrency/creating-and-starting-threads.html>

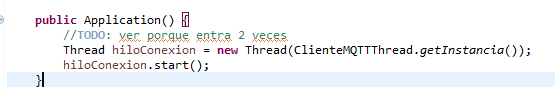
Autenticación SSL:

<https://es.wikipedia.org/wiki/Transport_Layer_Security>

Java KeyStore:

<https://docs.oracle.com/javase/7/docs/api/java/security/KeyStore.html>

Modificamos el constructor de la clase main (Application) para iniciar el hilo al momento de instanciarse la app:



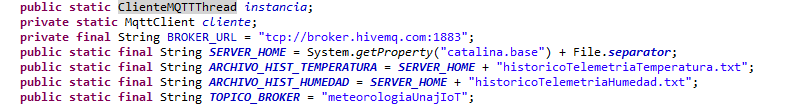
\*\*el hilo es la implementación de un Singleton

<https://msdn.microsoft.com/es-es/library/bb972272.aspx>

\*\*\*Aclaración: en nuestro caso utilizamos la implementación de Spring boot para hilos:

<https://egkatzioura.com/2017/10/25/spring-and-threads-taskexecutor/>

En el Hilo configuramos la parametría para la APP:



BROKER\_URL = URL del broker MQTT.

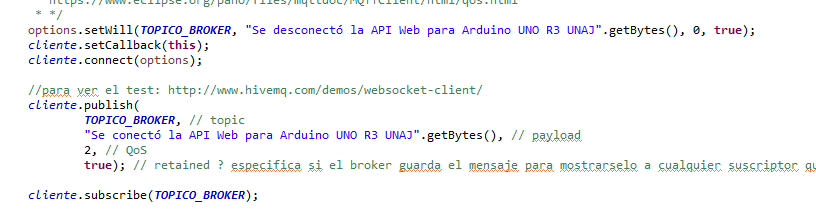
SERVER\_HOME = la ruta de directorio del servidor de aplicaciones que estamos usando.

ARCHIVO\_HIST\_TEMPERATURA = la ruta de directorio del archivo donde se guardan las publicaciones de temperatura.

ARCHIVO\_HIST\_HUMEDAD = la ruta de directorio del archivo donde se guardan las publicaciones de humedad.

TOPICO\_BROKER = el tema al cual publicamos y suscribimos para este proyecto.

Basicamente:

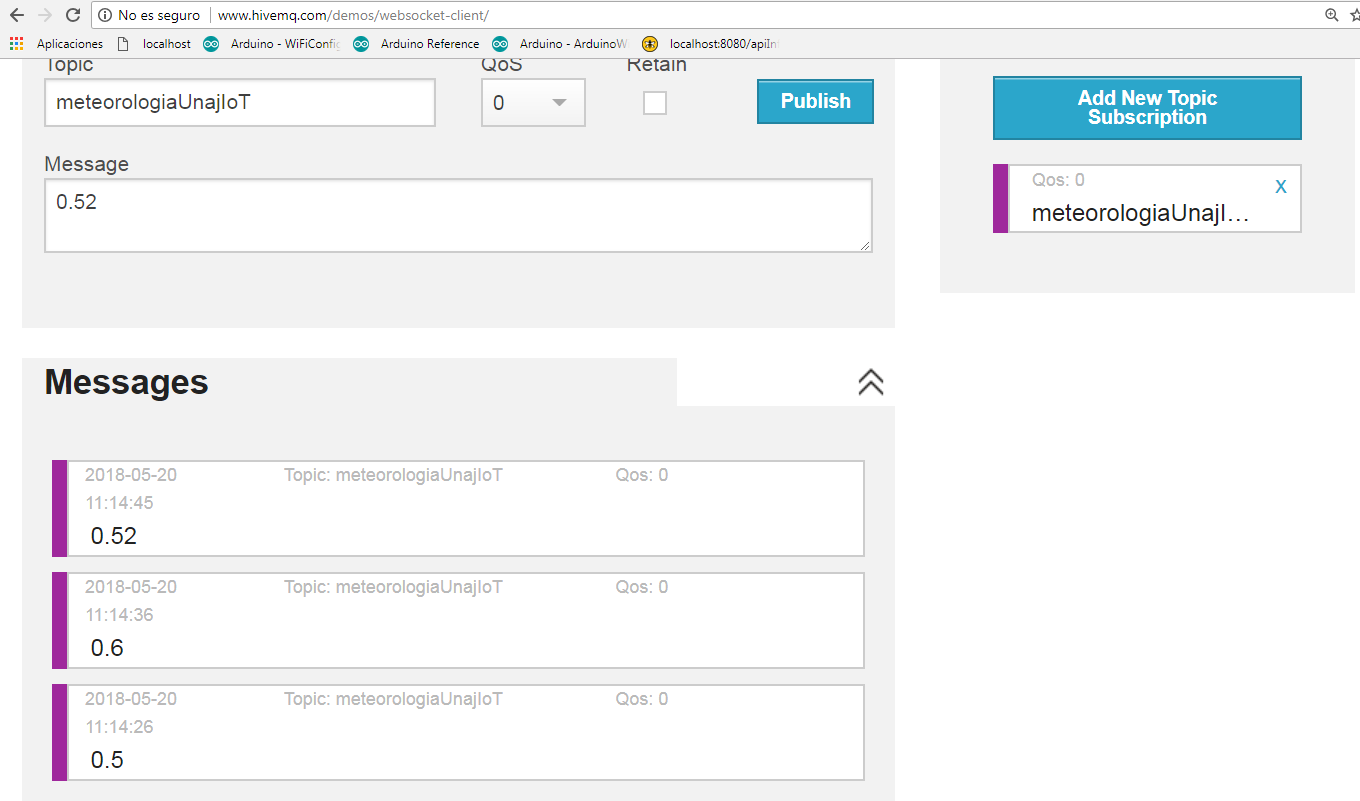


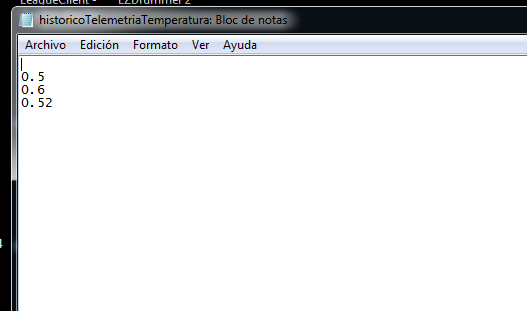
* *Al momento de conectarnos publicamos un mensaje para cualquiera que se suscriba.*
* *Luego cuando revocamos la suscripcion al Broker nos llega el mensaje de desconexión.*

En nuestro caso, como nuestra API está suscripta al mismo tópico, nos llega el mensaje de conexión y se ejecuta el siguiente código:

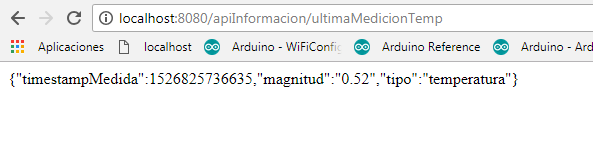
el cual escribe en un archivo en el directorio del servidor de aplicaciones.

Finalmente:

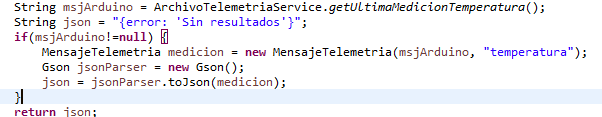




/ultimaMedicionTemp



El la URL /ultimaMedicionTemp es interceptada por el método ultimaMedicionTemp():

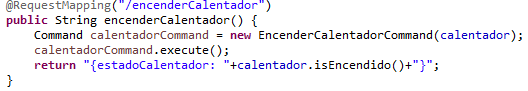


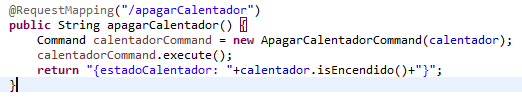
El cual lee el archivo de temperatura y trae la última línea con un servicio:



La url /encenderCalentador

Es interceptada por el método del mismo nombre:



Al igual que el /apagarCalentador: 

Estos dos últimos métodos, utilizan la implementación del patrón de diseño “Command” para controlar el estado del Calentador (Estufa o lo algún dispositivo calentador).

SEGURIDAD:

Conexión por socket con conexión SSL al broker desde la API:

Tutorial a seguir:

<https://www.thomasvitale.com/https-spring-boot-ssl-certificate/>

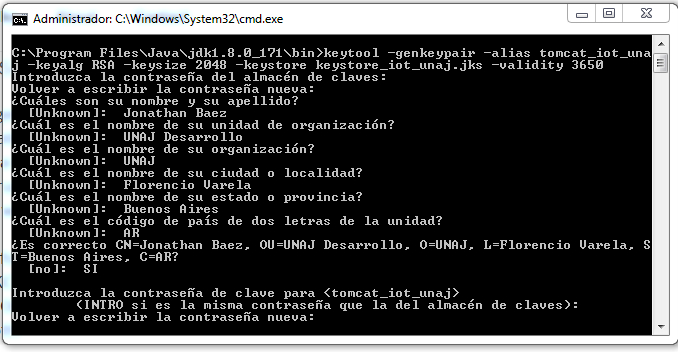
Generamos un certificado legitimado por la JVM.

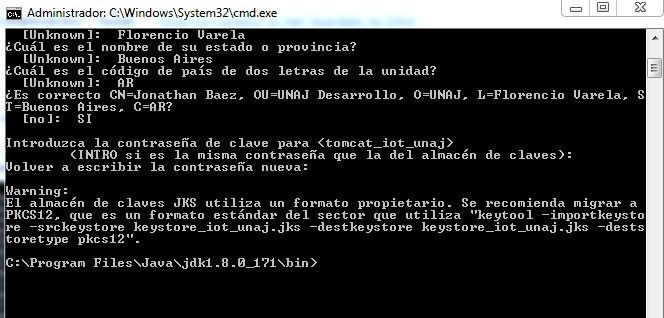
1- Generamos el store de la JVM de Java para guardar los certificados:

Y ejecutamos la siguiente sentencia en el home de Java (JDK)

keytool -genkeypair -alias tomcat -keyalg RSA -keysize 2048 -keystore keystore.jks -validity 3650

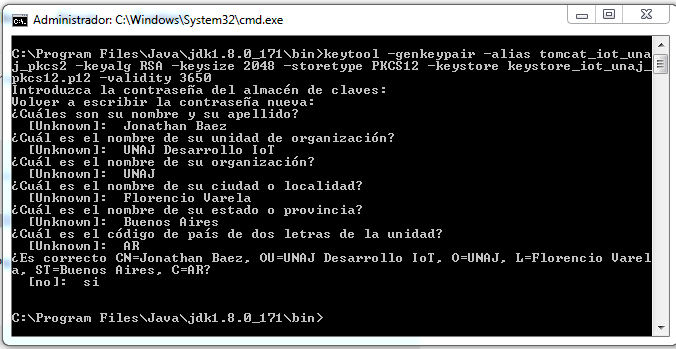
Y nos realiza preguntas de autenticación:





Nos da una advertencia del tipo de encriptación del archivo, por lo tanto generamos un keytore con un método de encriptación más sofisticado:

<https://es.wikipedia.org/wiki/PKCS>



Vemos cómo se generó el certificado: keytool -list -v -keystore keystore\_iot\_unaj\_pkcs12.p12



Es relevante notar que cuando se crea con el algoritmo de encriptación de la JDK por defecto la extensión del certificado es .jks y cuando se crea con PKCS es p12. (en este caso PKCS12).

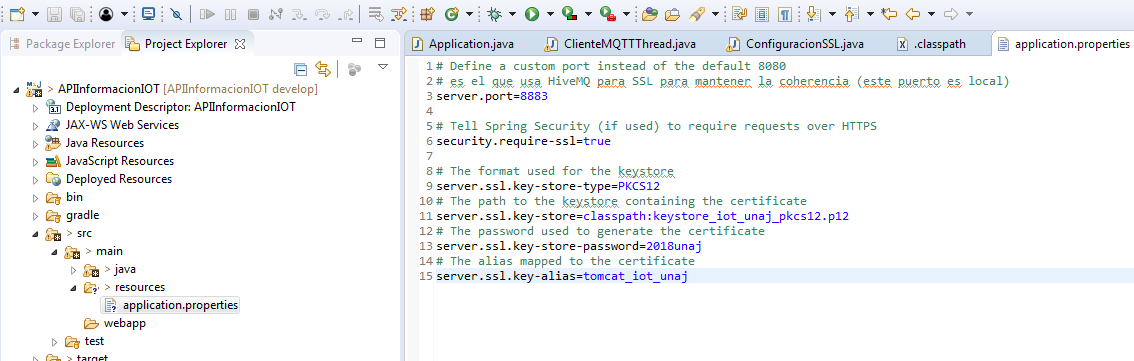
//TODO BUSCAR DIFERENCIAS ENTRE JKS Y PKCS

Ahora, agregamos a SpringBoot un archivo de configuración para habilitar la conexión HTTPS en el Tomcat:

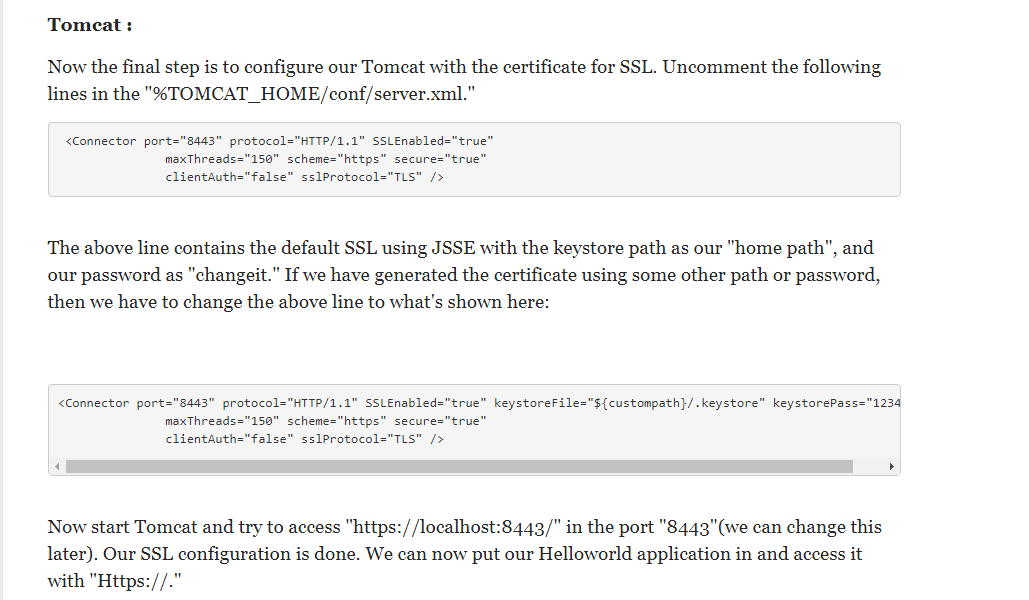
Orden de carga de properties de Springboot:

<https://stackoverflow.com/questions/25855795/spring-boot-and-multiple-external-configuration-files?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa>

Agregamos la configuración del certificado SSL en Spring boot:



Pasamos a configurar la conexión del Tomcat mediante HTTPS:

En caso de querer configurarla directamente desde el servidor, se agregar la linea de abajo en el archivo server.xml del directorio del Tomcat. 

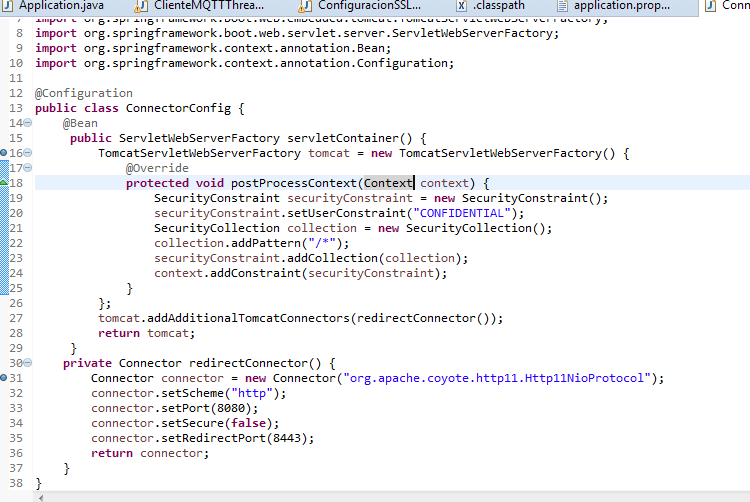
<https://dzone.com/articles/https-ssl-tomcat-jetty-jboss>

Para hacerlo más robusto, nosotros vamos a configurar dicha conexión programáticamente:

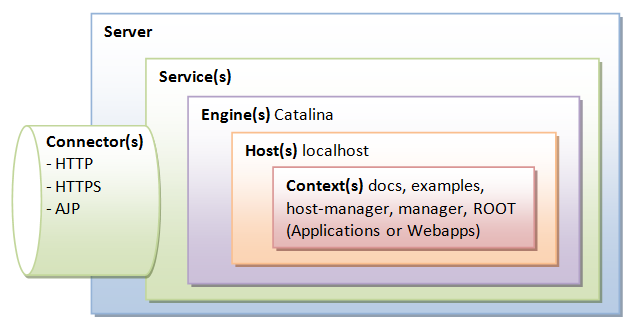
\*Aclaración: Esta solución aplica únicamente para Tomcat, para el resto de distribuciones de servidores existen otras posibilidades que no se abordarán en el documento.

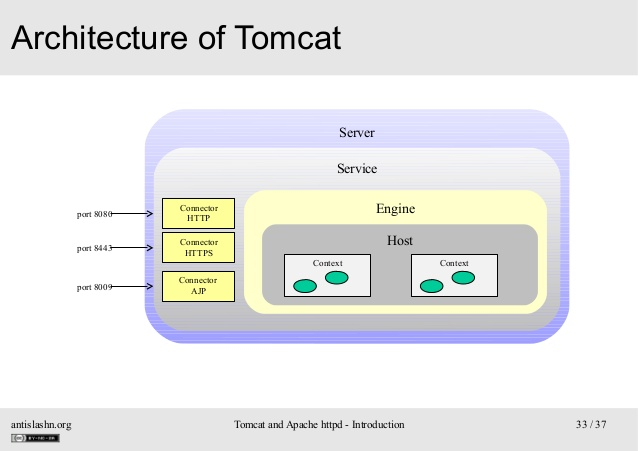
<https://stackoverflow.com/questions/47700115/tomcatembeddedservletcontainerfactory-is-missing-in-spring-boot-2?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa>

Agregamos una clase que configura el Tomcat para interceptar las request (sacado del link de arriba):

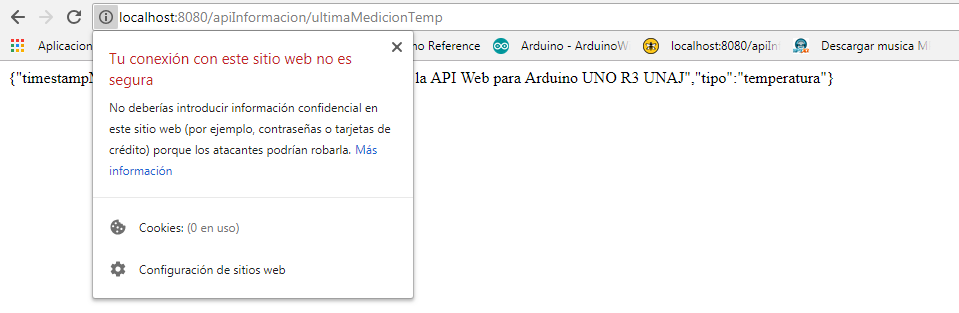


Spring Boot utilizará el bean ServletWebServerFactory el cual es configurado por el método servletContainer.



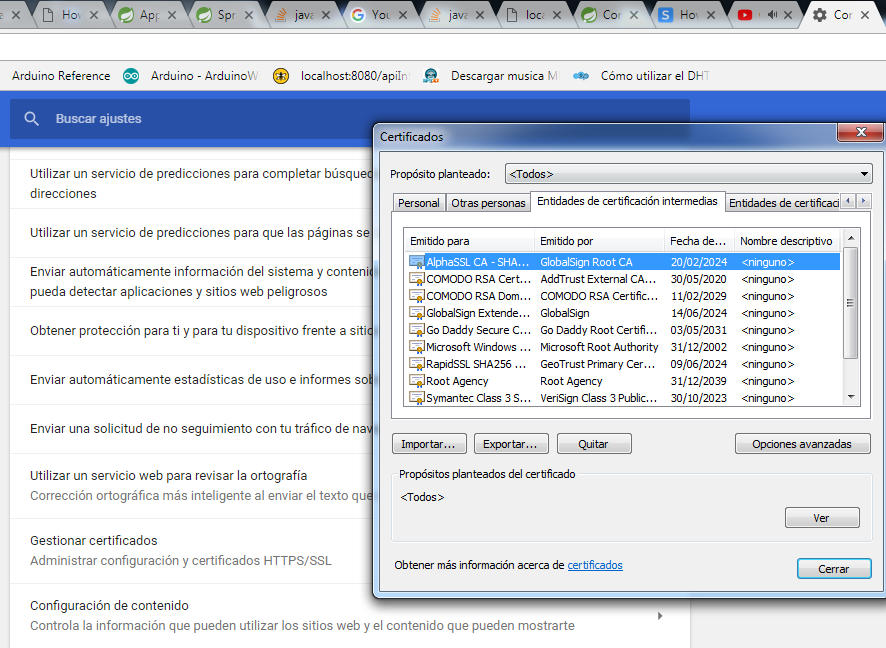


Ahora realizando una consulta a la API, vemos que el Browser no puede validar la identidad del Web Server. Ya que como utilizamos un certificado auto validado o legitimado por nuestra JVM no puede reconocerlo como proveniente de una entidad legitimadora.



Se puede agregar el certificado manualmente:

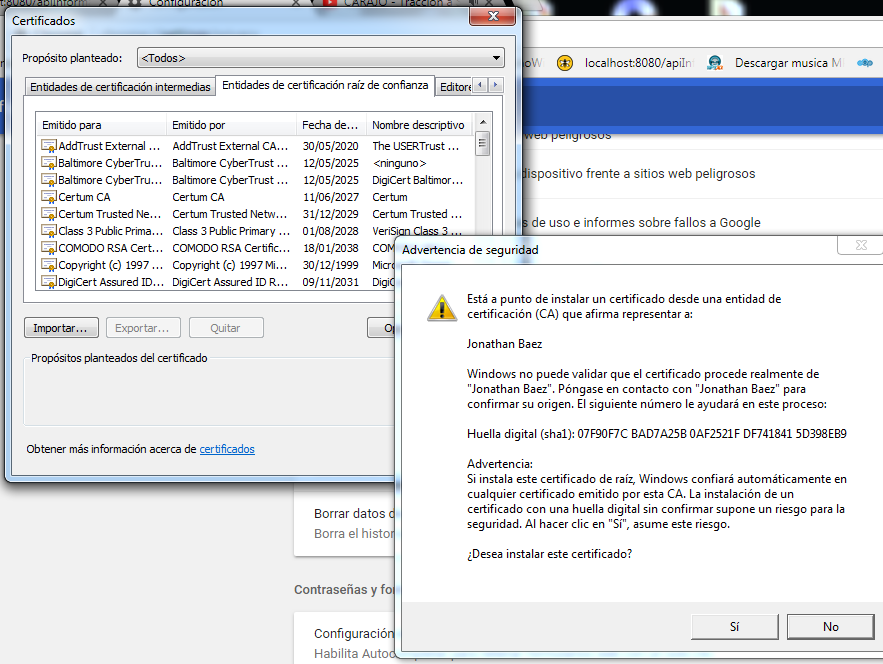
<https://support.securly.com/hc/en-us/articles/206081828-How-to-manually-install-the-Securly-SSL-certificate-in-Chrome>

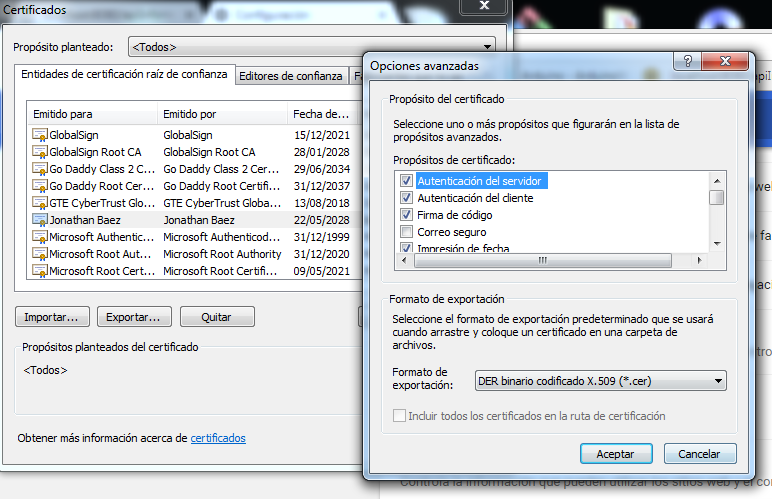


Como vemos, existen entidades certificadoras en las cuale el cliente confía.

Agregamos el nuestro:

Entidades de certificación raíz de confianza ---> Importar y completar la fase de autenticación





<https://howtodoinjava.com/spring/spring-boot/spring-boot-ssl-https-example/>

<https://www.ibm.com/support/knowledgecenter/es/SSFKSJ_7.1.0/com.ibm.mq.doc/tt60316_.htm>

Trust Store y Key Store:

<https://stackoverflow.com/questions/6340918/trust-store-vs-key-store-creating-with-keytool>

<https://github.com/michaelklishin/mqtt-tls-playground/blob/master/java-paho/src/main/java/com/novemberain/examples/mqtt/TLSConnection.java>