LIFERAY IOT PLUGIN

Documentation

Liferay IoT Plugin

A guide trough design concept, implementation and use of LIoT plugin

Aristide Cittadino
aristide.cittadino@acsoftware.it

Introduction

Scope of this document is to present the Liferay IoT plugin.
First we will introduce the design concept and then we'll go deeper, analyzing the implementation and how to use it.

The idea behind this project is to transform liferay in a data collector exploiting its core functionalities such as message bus allowing to connect it to the major streaming technologies or queues.
The other crucial idea is to transform liferay (the control panel) in order to control a Big Data engine such Apache Spark.
From the control panel you can:

- Insert a new connection to a broker: specifying properties such as topics, or custom properties. The available brokers are:
  - Mqtt
  - Kafka
- Define a Broker Listener, a way to decide where data coming from broker must to be sent. At now the only broker listener is Message Bus. This means that at the beginning of your project you can connect liferay to a broker and receive directly on the MESSAGE BUS all data coming from it.
- Setup Apache Spark by loading jars, schedule jobs on the fly, or schedule jobs on events or dates.
- Use the internal API exposed by REST services to raise events (and fire spark jobs) or manage data inside the platform.

Those are few things that you can do with this plugin, but this is the starting point, we hope that with you help we can increase the functionalities and transform liferay in a IoT platform.

Environment Setup

First of all, before we go deeper, let's introduce which technologies we have used:

- Liferay 6.2 ce GA 4 (updating tomcat to a version supporting web socket)
- Mosquitto Broker
- Apache Kafka 0.8.2.1
- Apache spark 1.3.1 built for Hadoop 2.4
- Apache spark job server 0.5.2

The first two technologies are the brokers, the second one is Spark with a Job Server that allows to control it by invoking some rest services.
Keep track of all technical info (addresses and port) of spark and brokers because we will put them later inside liferay configuration.

Installation

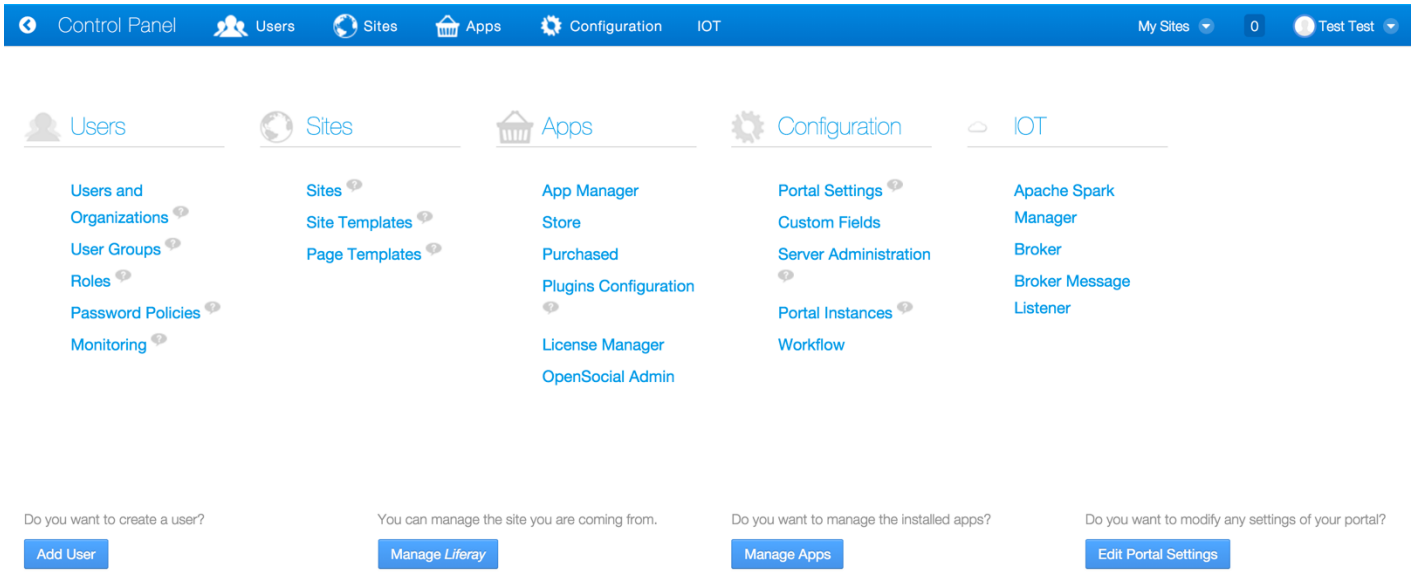
The LIoT plugin comes with 2 portlet projects and 1 plugin ext:

- LIoT-ext: is very simple, its scope is to add a new section inside the control panel called "IoT"
- ApacheSparkManager-portlet: is the portlet that interacts with Apache Spark Job Server
- LioT-Broker: is the portlet in which you can define brokers and how to dispatch messages.

If you deploy all plugins at once, and after the deploy receive some exceptions please restart liferay.

This project requires, for some functionalities (broker log), websocket libs, so if your container doesn't support websockets please update it or install a new container supporting them. Nevertheless, this is an important feature you can still run the plugin without web sockets support, some real-time functionalities such as: broker real-time view will not work.

Once deployed all plugins, we should see something like this:



As you can see the last section is “IOT”.

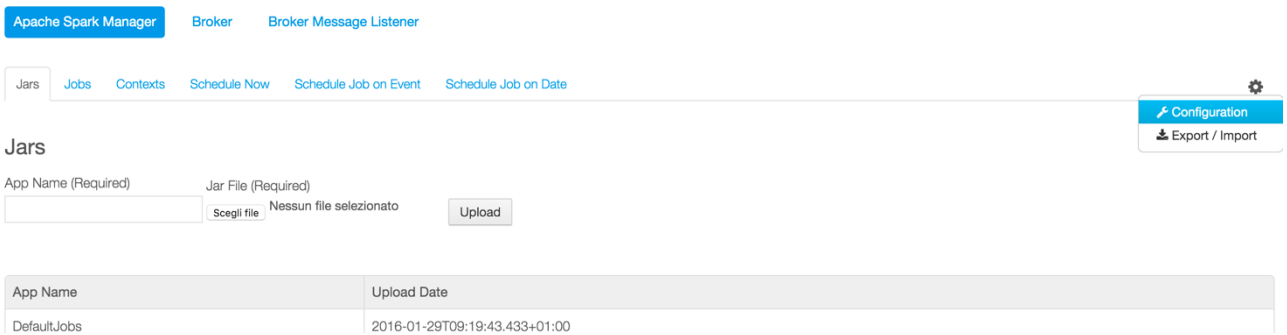
All settings are made by the control panel portlet settings, not portal-ext.properties.
Let’s go analyzing all the functionalities!

Apache spark Manager

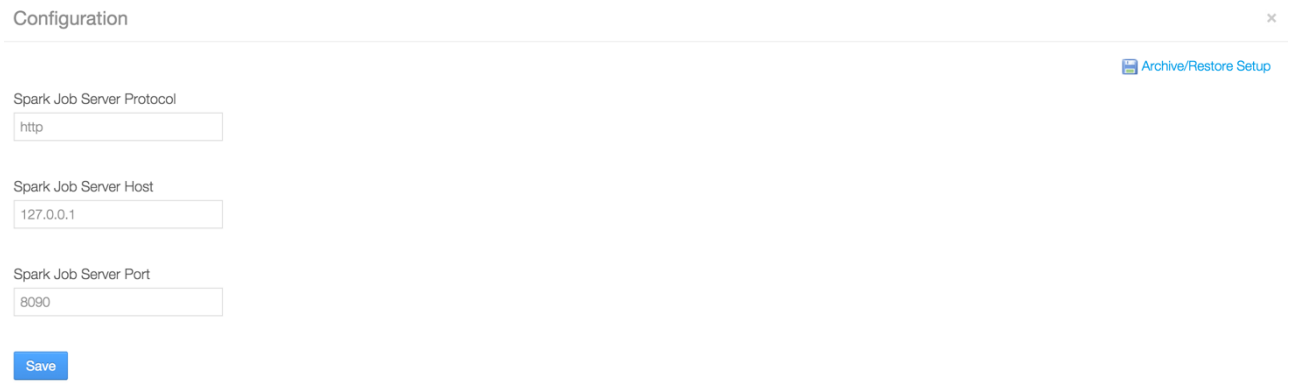
The goal of this portlet is simple, managing apache spark from Liferay in terms of:

- Uploading new jobs
- Check all ran job
- Check job contexts
- Schedule job on the fly
- Schedule job on date (based on cron expressions)
- Schedule job by events

First of all, we have to configure the portlet telling “where is” the spark job server. To do this go inside the Apache spark portlet and click the configuration button



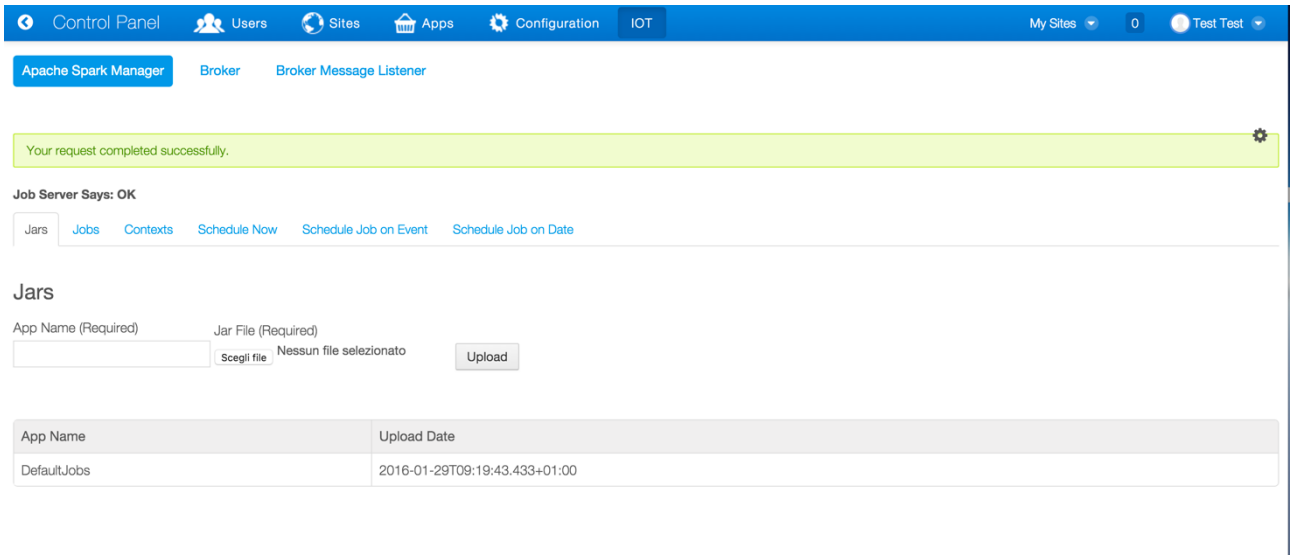
Then insert the info about spark Job server:



Keep in mind that you have to give spark Job server connection details not those of spark server.

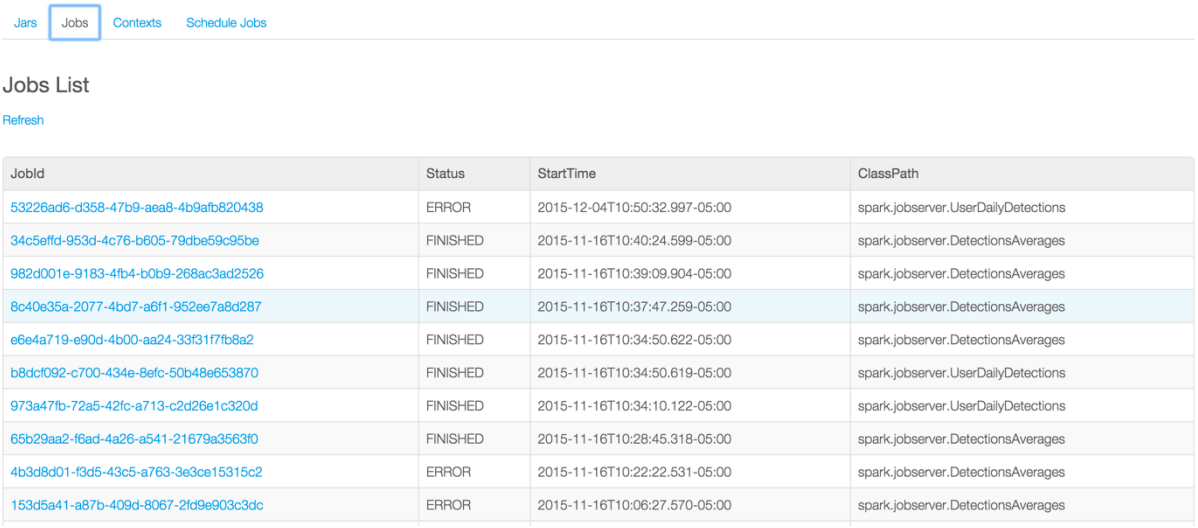
Uploading a Job to Spark

Uploading a jar file is very simple, just give a name and the jar and press “Upload” button.
Once jar is loaded it will appear on the jar list as below:



Jobs Monitoring

Clicking on “Jobs” tab you will see all scheduled jobs:

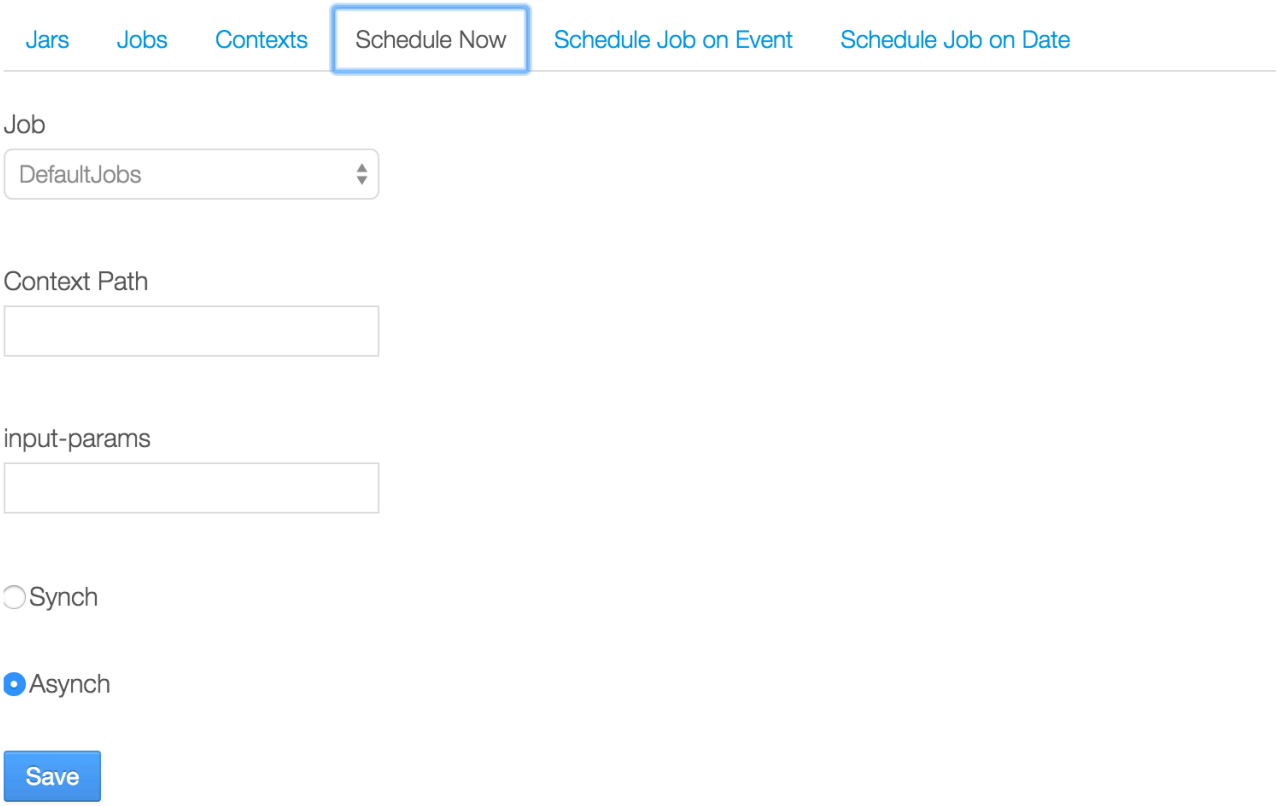


Clicking on jobId you can see the output of the job or the execution errors.

The Context Tab is for managing sparkJobServer contexts. Please read the spark and spark job server documentation for details.

Scheduling Management

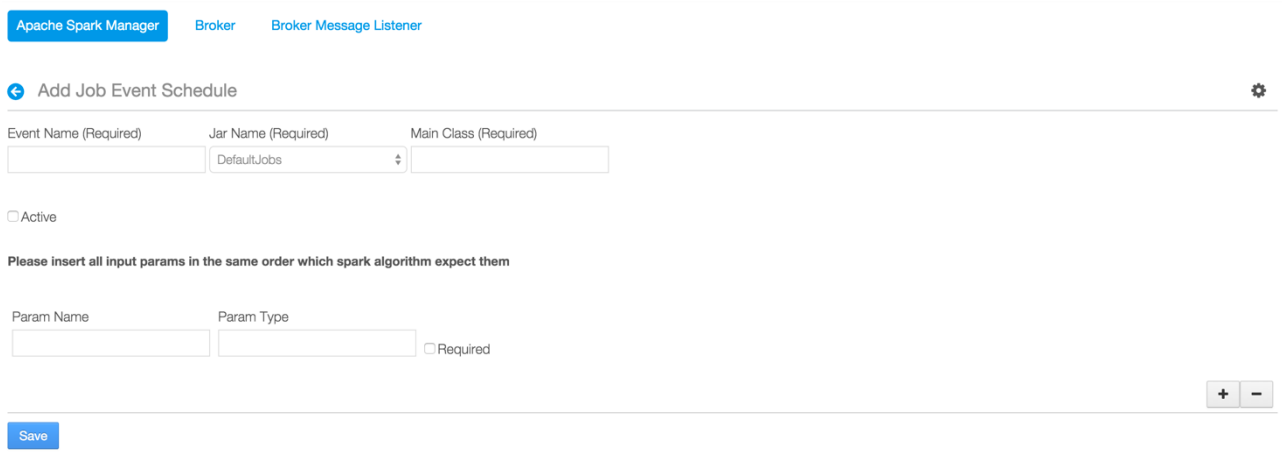
Let's see now the on the fly scheduler:



First select the uploaded jar the you have to specify:

- Main class with the path ex. It.acsoftware.sparkExample.MainClass.scala
- Input params
- Synch job it means that while the job is running your request keep waiting so you will be redirected after the job is finished
- Asynch you will be redirected as soon as you start the job and the you can check the job status in the Jobs Panel

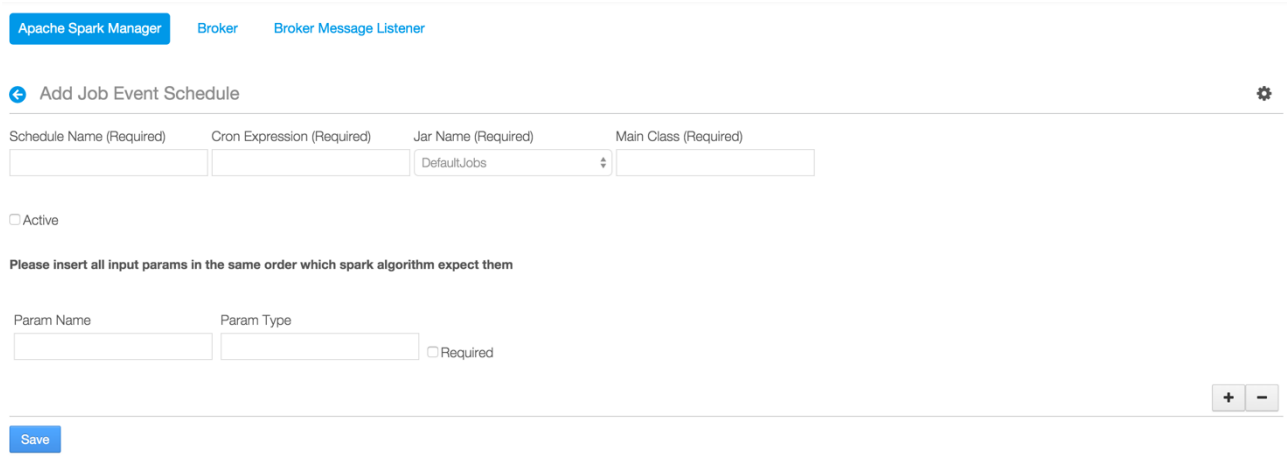
Scheduling events is a simple as on the fly scheduling. The main difference is that the job is not executed as you save the scheduling but it is executed when the defined event is “raised” inside the liferay platform.
This mechanism is very useful because you can start spark job at runtime whenever you need it.



Here we have:

- Event name: you will use it when you’ll tell the system to raise the event
- Jar name
- Main Class
- Active: defines if the job is active and so it can be run
- Parameters in the same order in which you load them in the spark job

For adding a date schedule you’ll have to specify the following parameters:



The same fields as event scheduling plus the cron expression. Please note that date scheduling is reloaded at startup by default, so once you save them you don't have to worry to reschedule them.

Using API

Good, this is the simple part in which you can easily upload jobs and configure the platform to execute them.
Now let's see how to control programmatically the spark portlet in order to Raise events or schedule job from java code.
You can include the **ApacheSparkManager-portlet-service** jar in your project and use the **ApacheSparkManagerScheduler** to

- Retrieve all events defined
- Raise an event
- Schedule date job

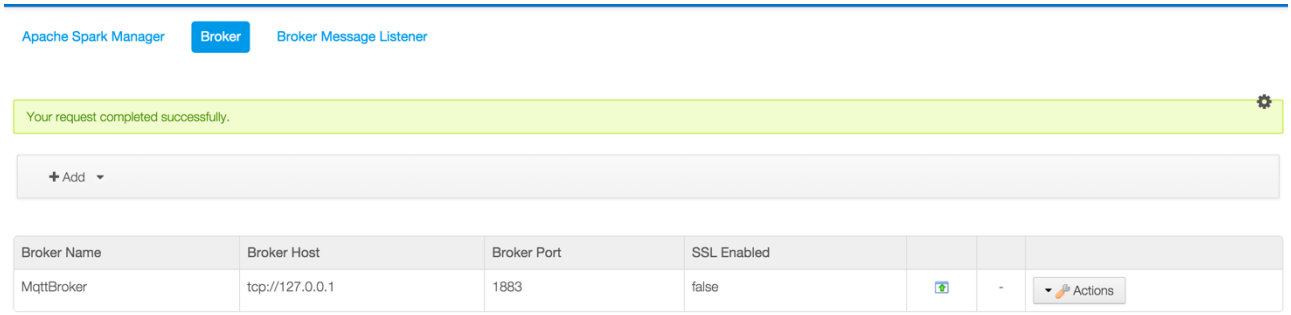
You can even use the REST service to communicate with this portlet. Please check the source code or API from /api/jsonws in liferay, to have more info or contact me!

Broker-portlet

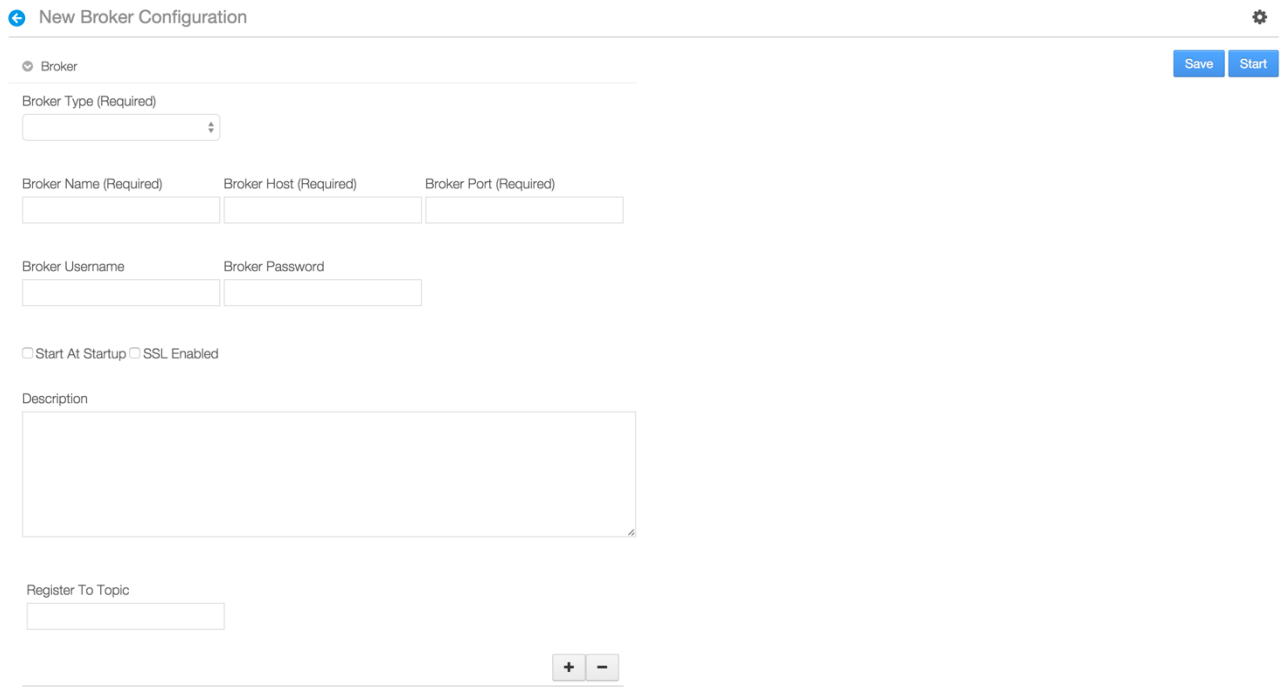
There are two main concepts behind this portlet:

- Let the user defines which brokers liferay must be connected to and don't worry about the communication but concentrate the development only to manipulating incoming data not to how to receive it
- Give to the developer an easy way to attach custom class to the incoming streams

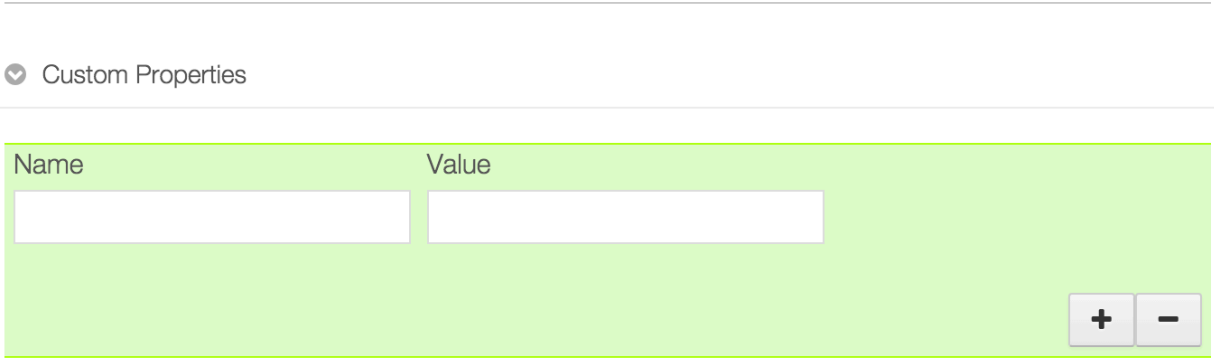
Let's start from the configuration point, go inside the broker portlet:



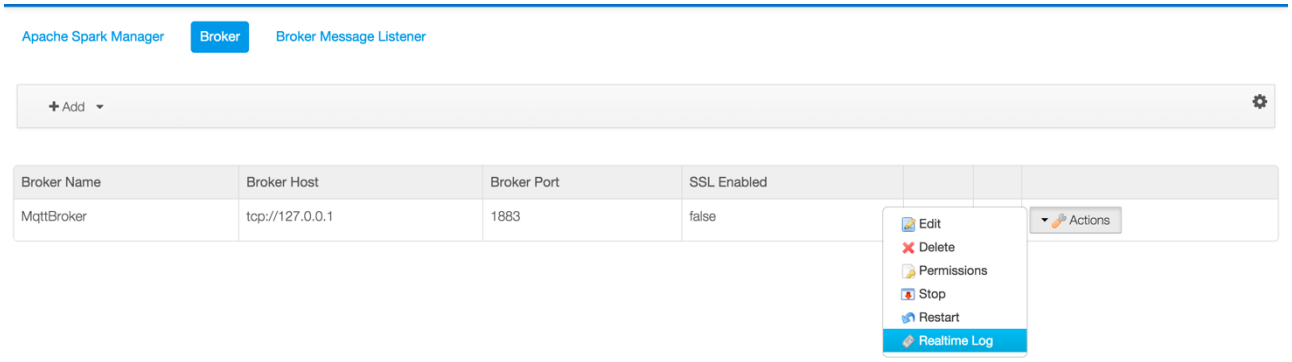
As you can see, i have defined one new broker connector called “MqttBroker” and it’s running (green up arrow). From the actions menu we can stop the service, restart it or have a real-time monitor. Let’s start to how to define a new broker connector.



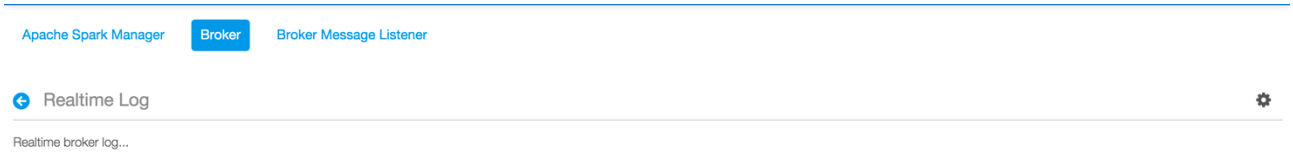
We Have different sessions: the first one is where we have to put normal info such as the broker type, name, host and so on.
You can even specify which topics (we can even use wildcards ex. myTopic/# or myTopic/+ if we use mqtt broker connector).
With auto-fields you can specify how many topics you want.
Each broker can have more custom properties.In the same page we have another section in which you can specify more properties:



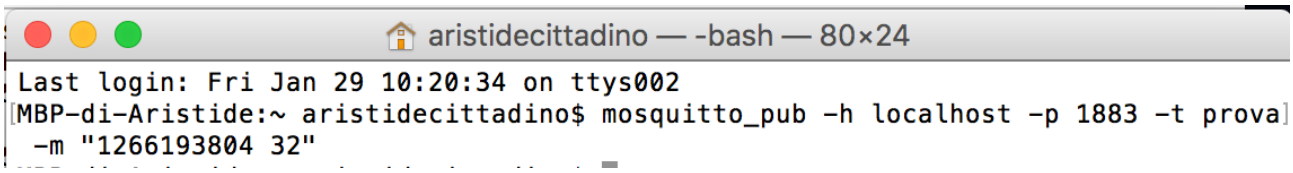
Once we have added our connector we can go the broker list and check if it's running.



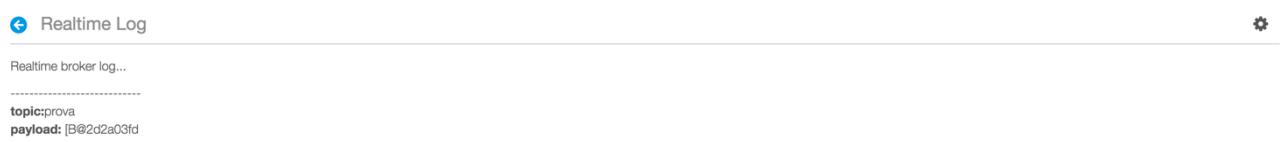
By Clicking on the real-time log we go in a websocket page directly connected to the broker



If you send a message (ex. from mosquitto) you will see it inside the same console without refreshing the page:



Sending a message on topic "prova" will result in seeing the message on the websocket page:



The payload is in byte[] then you have to convert in your model. We have use it in combination with ProtoBuf in order to serialize in an efficient way our custom model.

That's it! By adding a broker you can directly connect to it and receive messages.
There's something missing, how can we "attach" our class to incoming message streams? We can use liferay MessageBus!

This is very simple concept, go in the broker message listener portlet define a new message listener on your broker and attach it to the message bus. This portlet could be extended to support more kind of "Observing" mechanism. At now, the only supported is MessgeBus.
In the definition page you have to insert only the message bus destination and then setup your custom portlet to receive message on the same message bus topic. You can find some materials on the web on how setting up message bus communication in liferay.

You can even use REST services to control the broker connectors.

This is not enough, we can do more! You can even define your custom class extending **BrokerMessageListenerService** in your custom portlet and register it by passing it to the **BrokersServicesManager.register** method.
With this method you will bypass liferay message bus but your class will be invoked directly from the connector once message is arrived.
In order to use this functionalities you must insert the **LIoT-Broker-portlet-service.jar** in your project

Conclusion

This is the first version of this project but i think it can be very useful in IoT contexts. If you find bugs or have some suggestions please contact me aristide.cittadino@acsoftware.it

You can find some examples in the third portlet i have uploaded **LIoT-examples-portlet**.

See you!