InterConnect
2017

March 19–23
MGM Grand &
Mandalay Bay
Las Vegas, NV

IBM

Hands-on Lab
Session 6908
Streaming Analytics in IBM Bluemix using Message Hub and Apache Spark

Andrew Schofield, Event Services, IBM Cloud

You must include the first two pages of this template.

Table of Contents

Before you start 4

What you will learn 4

Part 1 – Setting up the Bluemix services 6

- Creating a Message Hub instance 6
- Creating an IoT Platform instance 9
- Connecting IoT Platform to Message Hub 10
- Creating an Object Storage instance 12
- Creating Object Storage Bridges in Message Hub 15
- Recap 15

Part 2 – Generating a stream of events to analyze 17

- Registering devices in IoT Platform for the simulated cars 17
- Generating events from the simulated cars 19
- Recap 19

Part 3 – Analysis of data in Object Storage 21

- Creating an Apache Spark instance 21
- Analysis using Data Science Experience 23
- Reading data from Object Storage 25
- More complex queries 26
- Visualization in the browser 26

Part 4 – Streaming analytics 28

- Preparing to connect to Message Hub 28
  - Uploading the Kafka 0.10.2 client JAR 28
  - Checking the classpath in Spark 29
- Streaming data into Apache Spark 30
- Streaming analysis in Apache Spark 32

What next? 34

Before you start

You will need a Bluemix account in the US South region. You can sign up for a 30-day trial here: https://console.ng.bluemix.net.

Also, you will need to sign up for Data Science Experience using your Bluemix account. You can sign up here: http://datascience.ibm.com.

You should now be ready to run the lab.

Lab user name: demo
Lab password: sample

There is a backup of the lab materials at `/labbackup.tar`.

Cloud services are continuously being updated. As a result, the screenshots in this document may not exactly match the services at the time you follow the instructions.

What you will learn

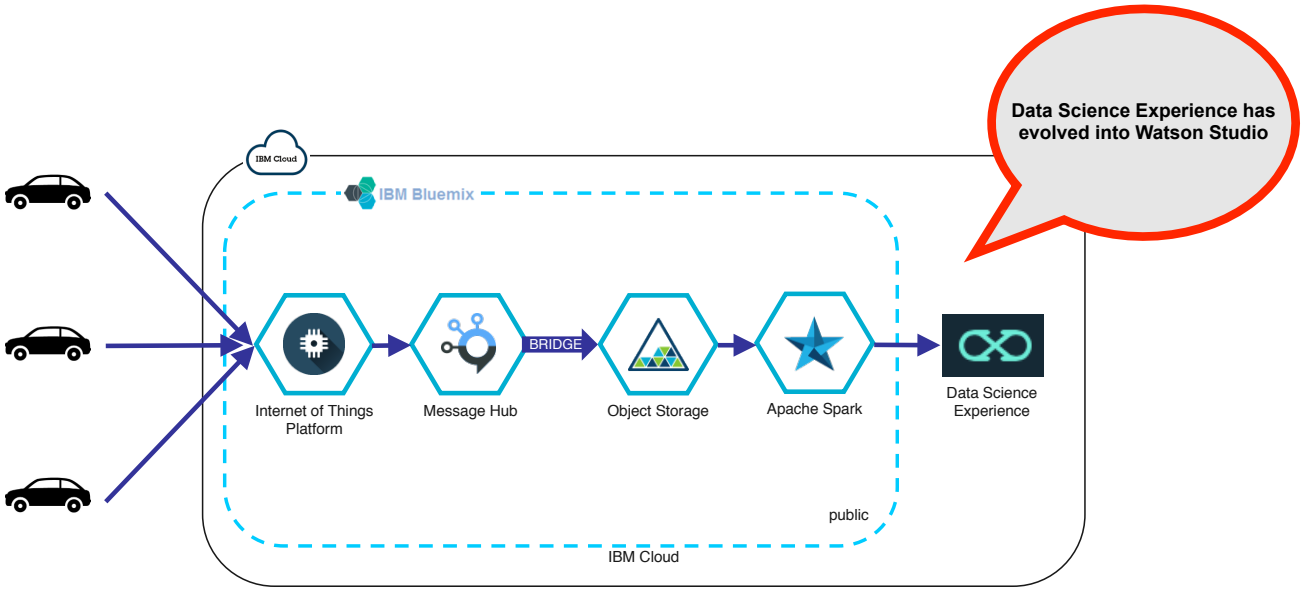
The combination of Message Hub and Apache Spark provides a powerful capability for performing analytics of streams of events, whether from IoT devices, financial trades or any other events in your business.

You will learn how to use the combination of Message Hub and the Apache Spark services in IBM Cloud to analyze events in a flexible, dynamic way.

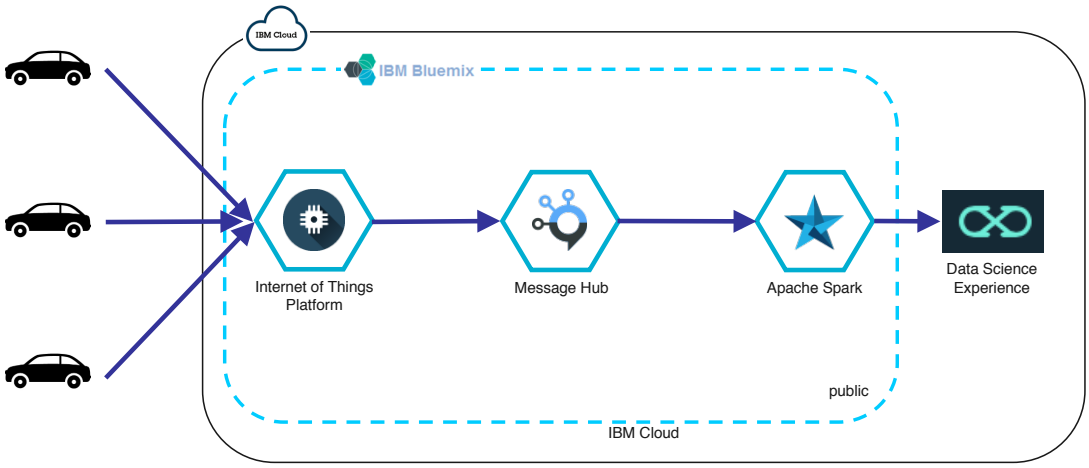
In the lab, an application running on your laptop simulates events from a fleet of 20 connected cars. You can perform analysis of the events easily in Data Science Experience using an interactive environment running in a web browser.

You will use Message Hub to collect the events from the simulated cars and feed them into the analytics service.

The lab shows two ways of running the analytics. By capturing the data in the Object Storage service, you can run analytics on data at rest. This way of working is useful if you want to process a static set of data repeatedly, or to restrict the analysis to a range of dates.



Alternatively, you can use Spark Streaming to process the stream of events directly out of Message Hub. This is known as streaming analytics.



Part 1 – Setting up the Bluemix services

We need three Bluemix services to get started:

- Watson Internet of Things Platform – to feed events from our simulated devices
- Message Hub – to buffer the events on their way to be processed by the analytics code
- Object Storage – to store the events for analysis

It's most convenient to create them in a different order: Message Hub, then IoT Platform, and finally, Object Storage.

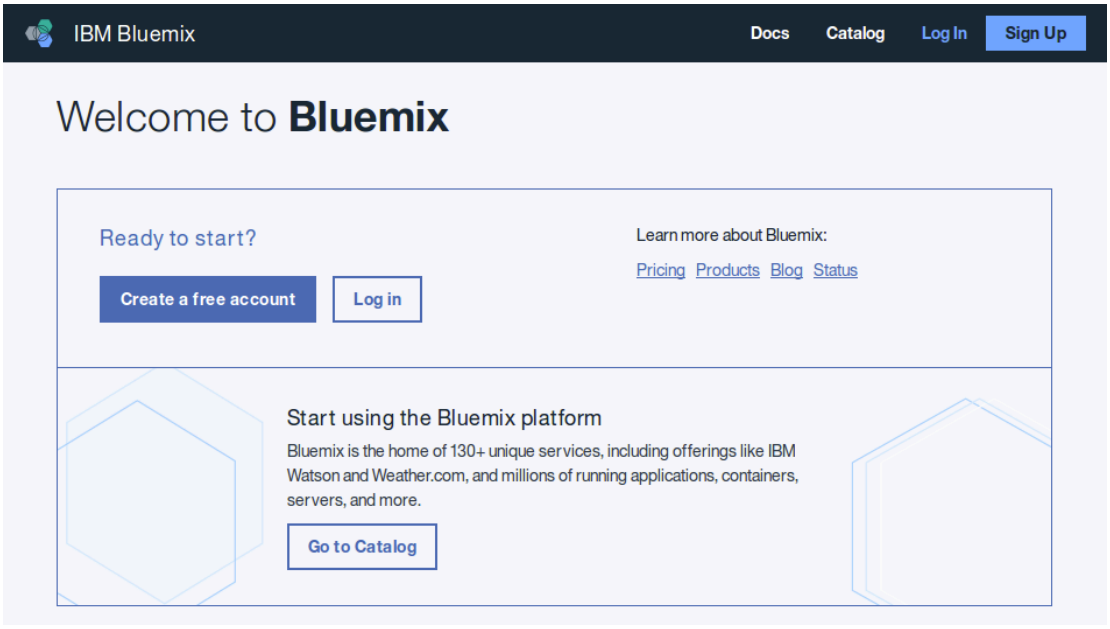
Creating a Message Hub instance

Open the Firefox browser from the launcher on the left-hand side of your screen.

In the bookmarks toolbar, click 

.

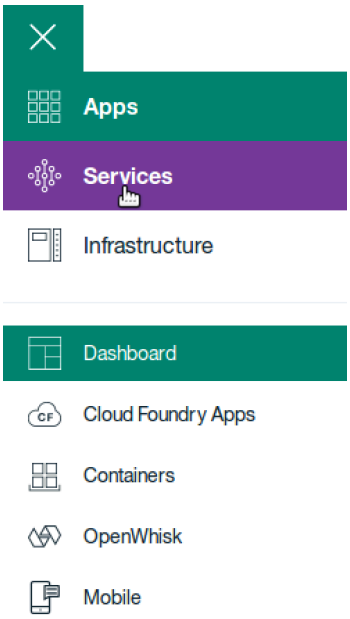
This takes you to the Bluemix home page at https://console.ng.bluemix.net which looks like this:



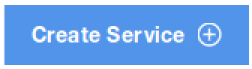
Log in to Bluemix with your IBM id which will take you to the Bluemix console.

In the top-left corner of the console, you'll see a menu icon like this:

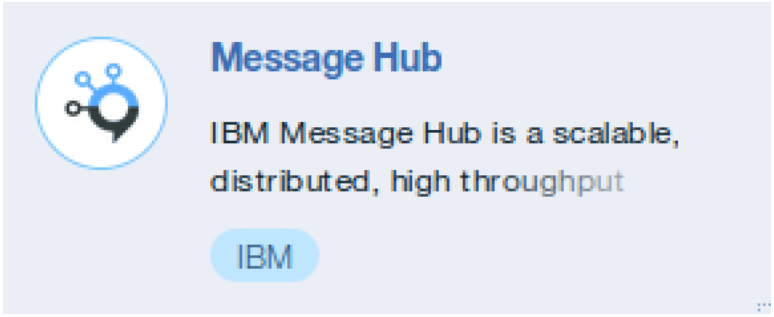
You can use this to change the perspective of the console. You use the services dashboard to manage the services (click Services and then Dashboard).



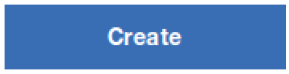
In the services dashboard, you'll see a button to create a service that looks like this:



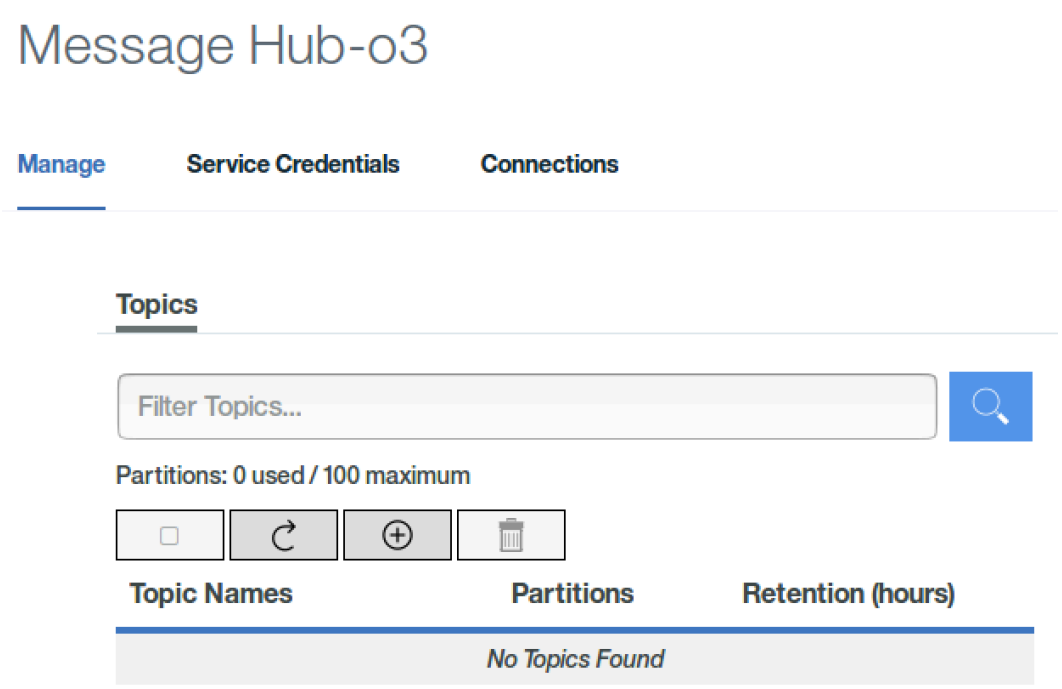
Press the button to take you to the services catalog. This contains the long list of services you can use in Bluemix. Take a moment to browse the list. You can filter the list down by typing into the Search box at the top. Find the Message Hub tile and click it:



The next page describes the service, how to use it and how much it costs. Click the Create button to create your service instance:



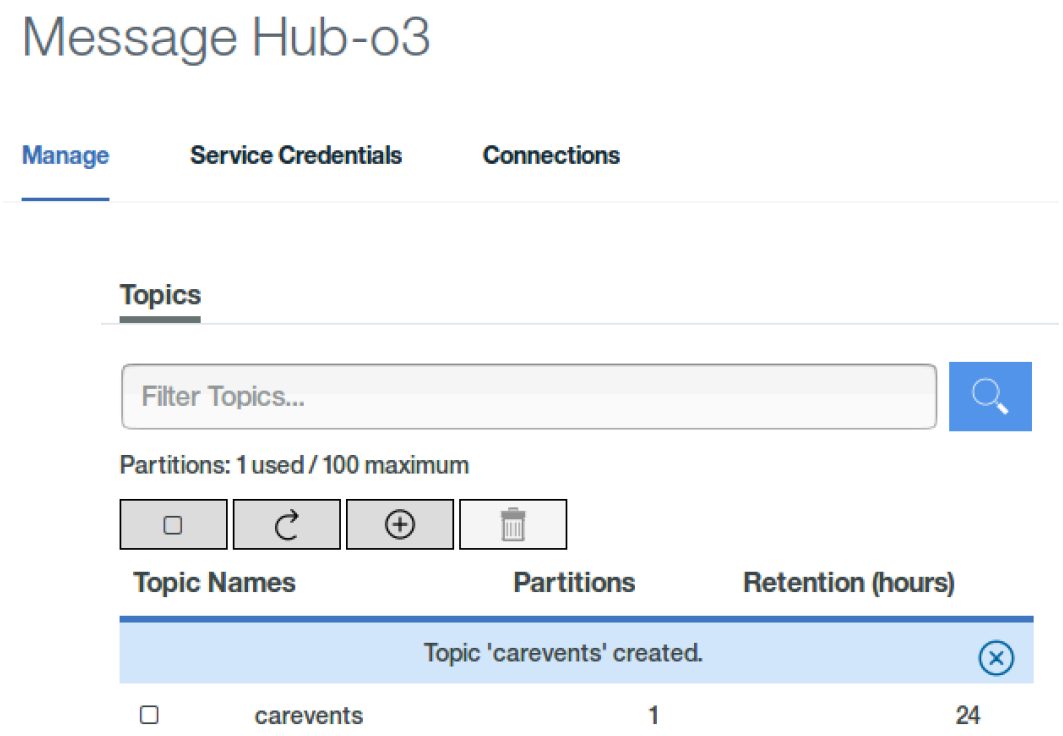
After a few seconds, you'll go to the Message Hub dashboard.



My service instance is called `Message Hub-o3`. Yours is probably slightly different.

Now create a topic to take the events from IoT Platform on the way to the analytics logic. Click the (+) button, and enter the name `carevents` for the topic name. Leave the other settings unchanged.

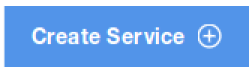
The dashboard should now look like this:



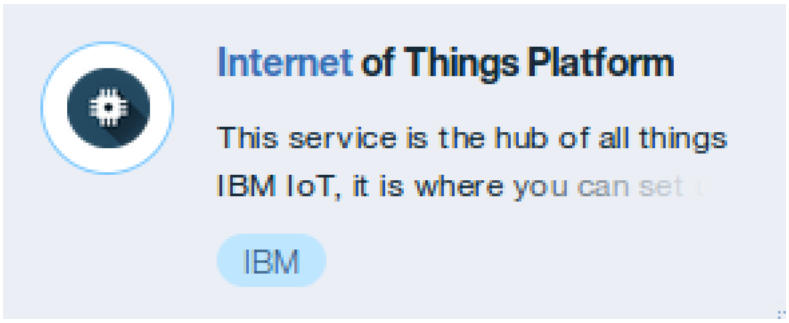
So, you now have an instance of Message Hub and have defined a topic called `carevents`.

Creating an IoT Platform instance

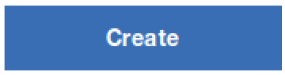
Go back to the services dashboard in the Bluemix console and begin the process of creating another service instance by clicking this button:



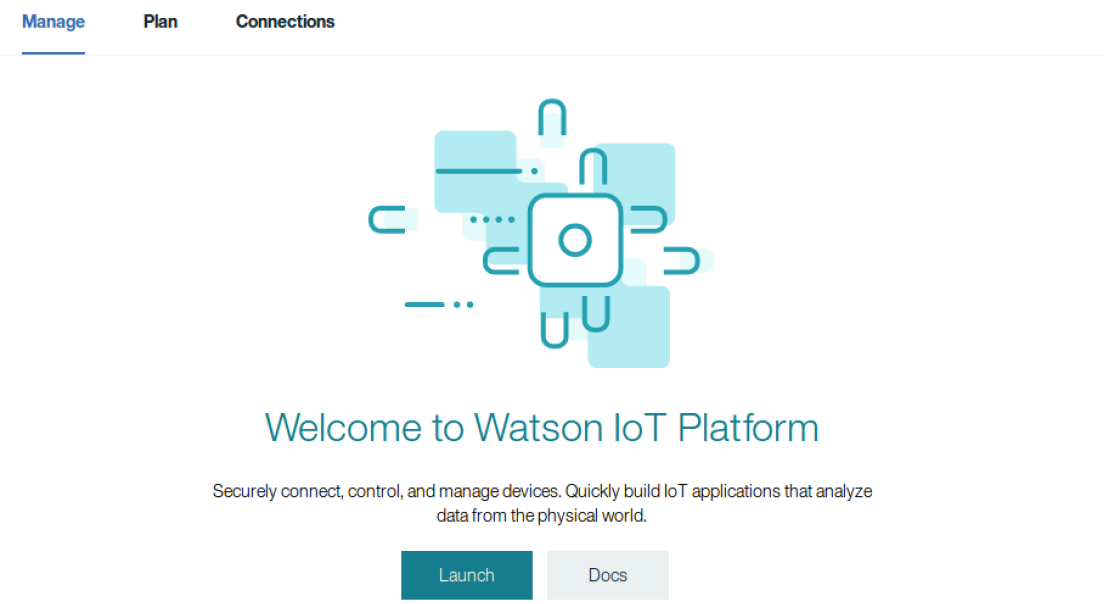
In the services catalog that you can now see, find the Internet of Things Platform tile and click it:



The next page describes the service, how to use it and how much it costs. Click the Create button to create your service instance:



After a few seconds, you'll go into the IoT Platform service UI.



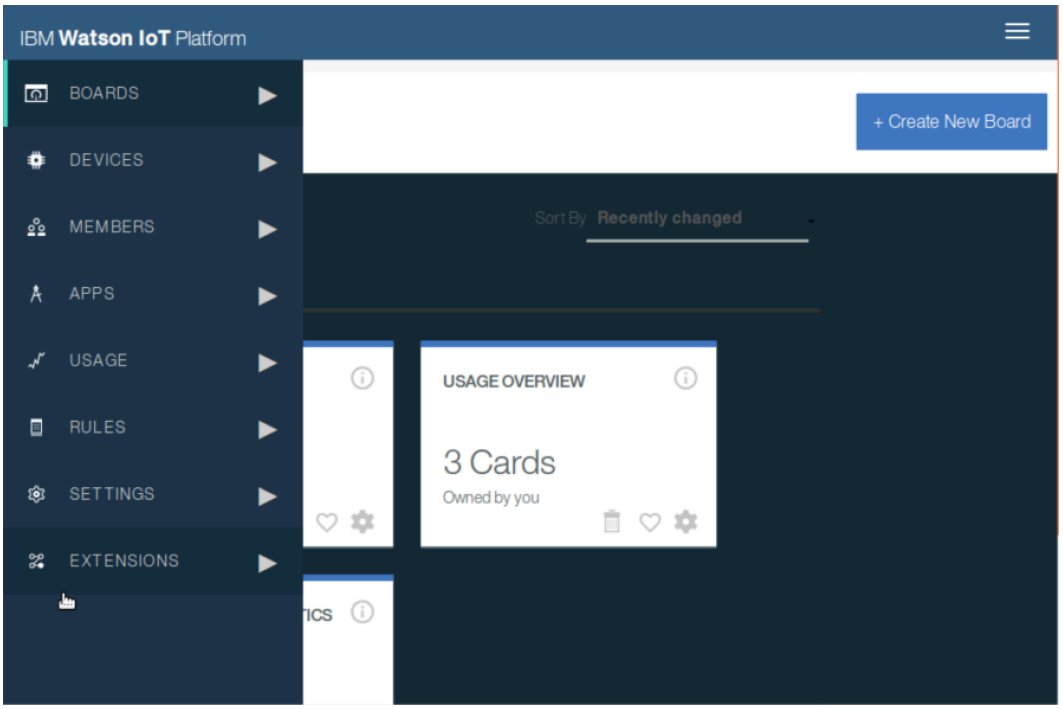
The dashboard for IoT Platform is outside the Bluemix console. Launch it now from the Bluemix console by clicking this button:

Launch

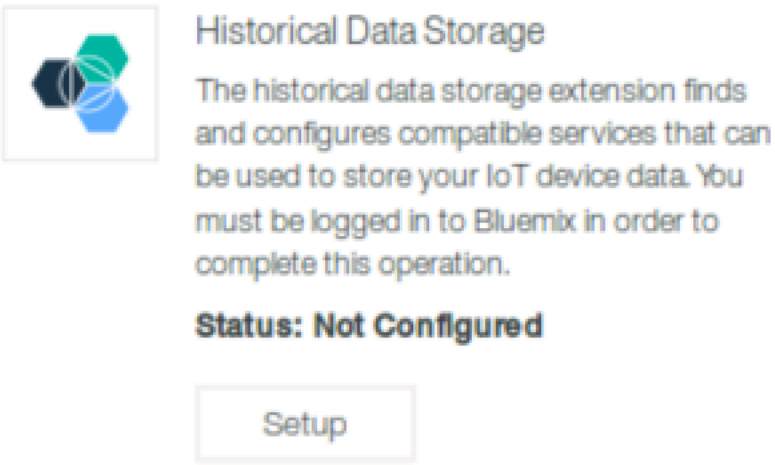
Connecting IoT Platform to Message Hub

You'll need to tell IoT Platform to send the events into Message Hub and grant it permission to do so.

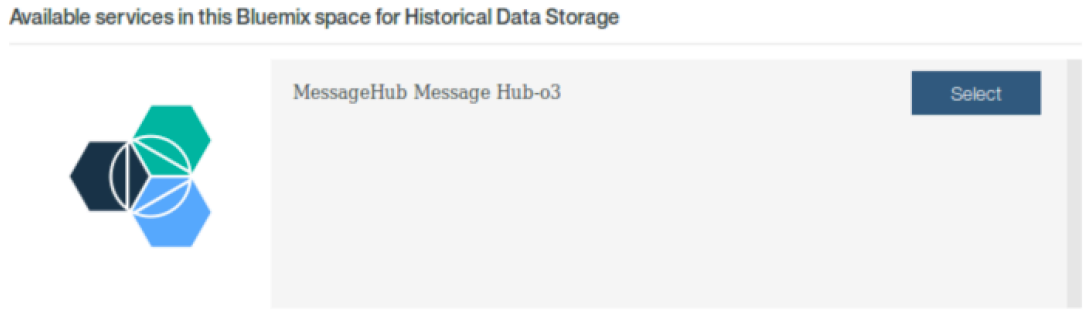
The IoT Platform dashboard looks like this:



In the menu, select EXTENSIONS and then under Historical Data Storage, click Setup.

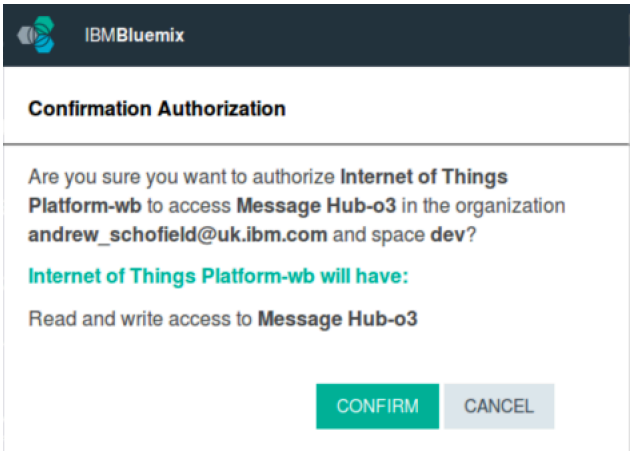


Message Hub can be used as a historical data storage extension and your Message Hub instance appears in the list of services.



Select the Message Hub instance. Enter the name of the topic you created earlier, `carevents`, as the default topic and click Done.

Now IoT Platform “binds” to Message Hub and it needs to ask your permission using a browser pop-up window. If you notice that Firefox blocks the pop-up window, permit it for this site. You’ll see a pop-up like this:

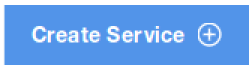


Click CONFIRM and you're done.

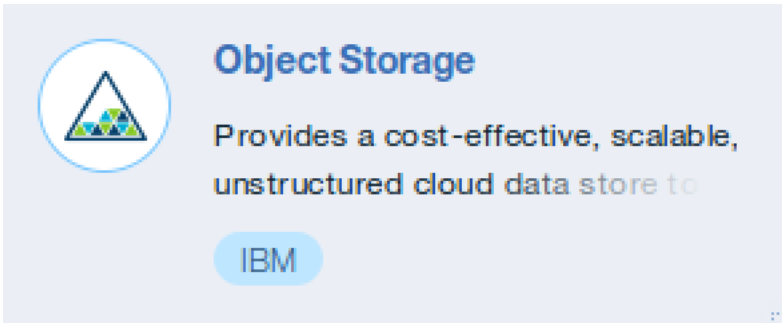
Creating an Object Storage instance

You need an instance of the Object Storage service for storing the events, and also because the Data Science Experience service needs somewhere to store its data.

Back in the Bluemix console, go back to the services dashboard and create another service instance by clicking this button:

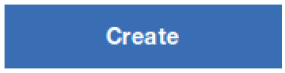


In the services catalog that you can now see, find the Object Storage tile and click it. Be careful to choose Object Storage and not Cloud Object Storage – S3 API.

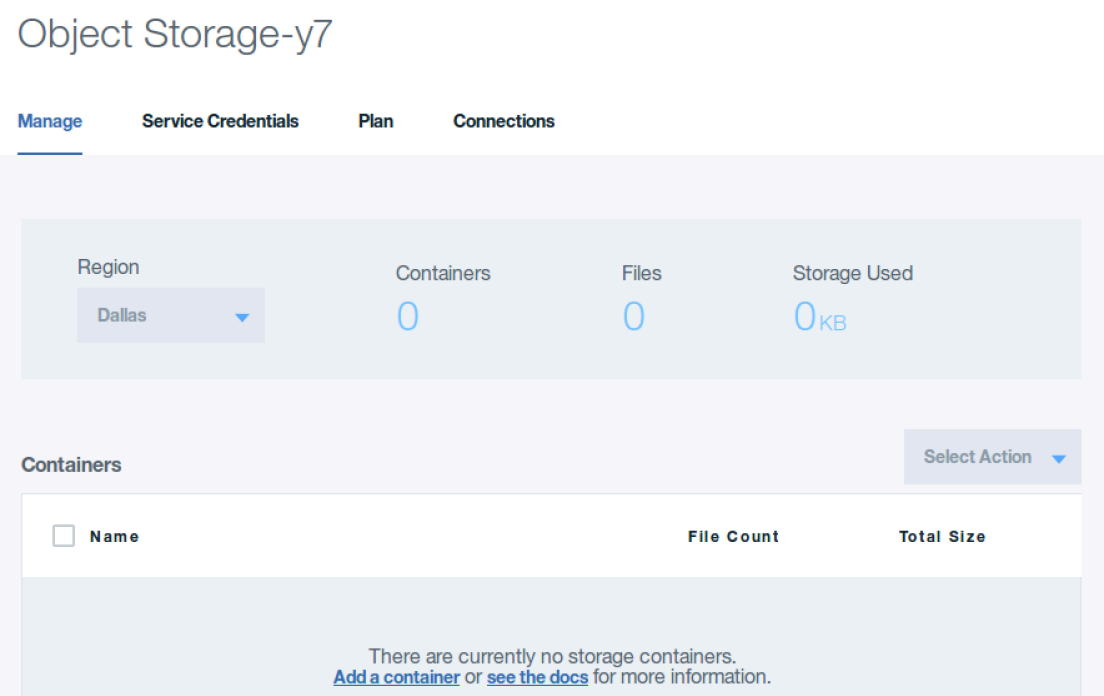


There are two different versions of IBM Cloud Object Storage. Clients can choose between a S3 and a SWIFT version. Please follow the "Corrected Instructions" document to create the Object Storage

The next page describes the service, how to use it and how much it cos
Click the Create button to create your service instance:

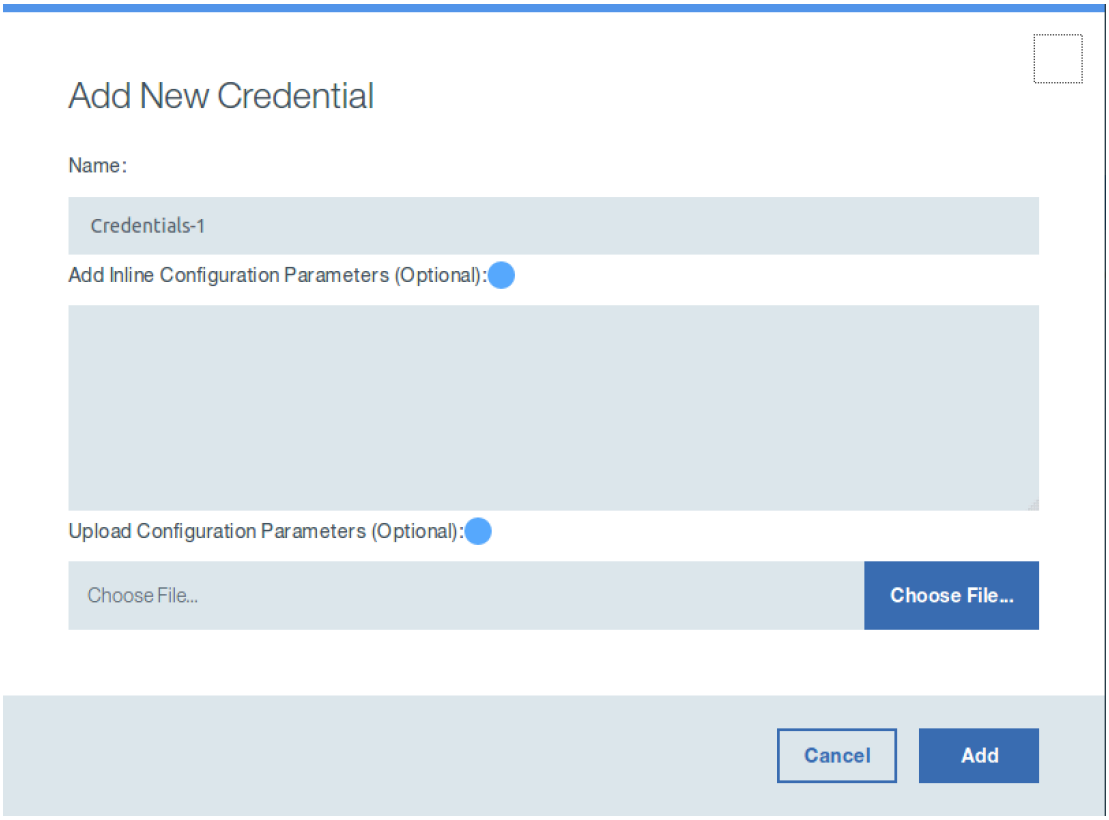


The creation process takes a few moments and you will be sent back to the Bluemix console where you will be able to see when it completes. Once the service instance has been created, you can go into the Object Storage service UI. Here you'll be able to see the data arriving from Message Hub, but for now it's empty.



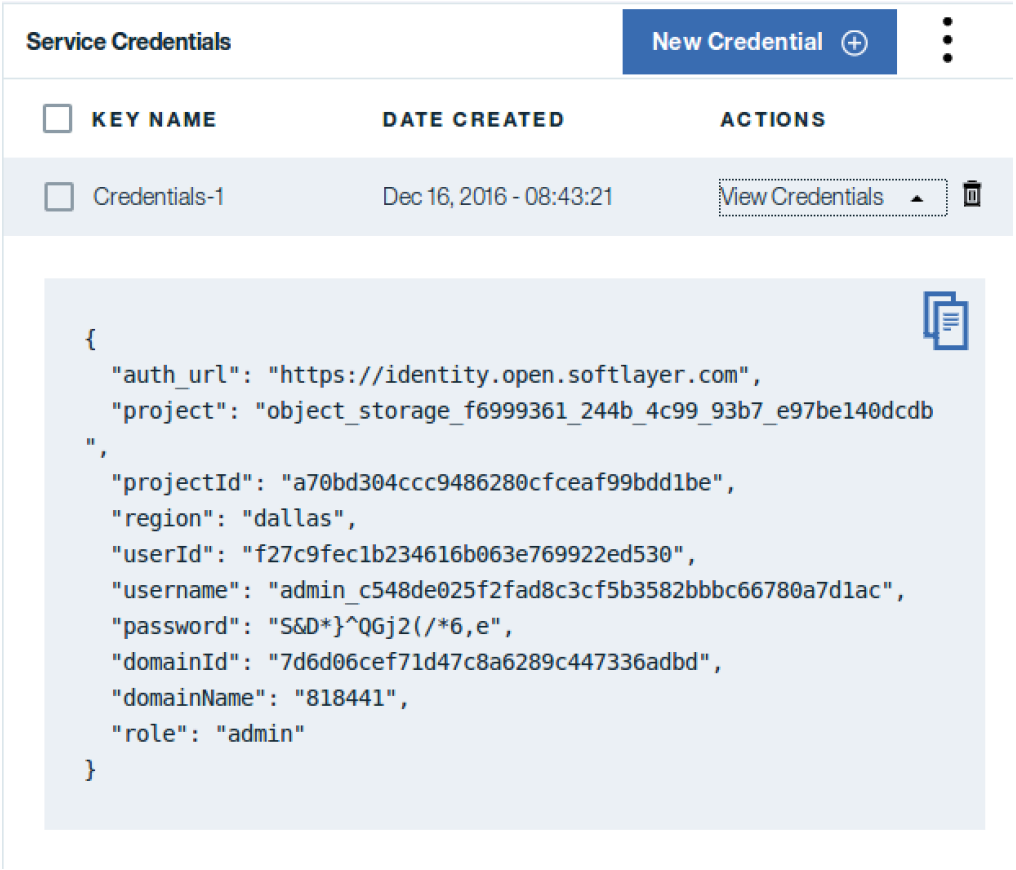
You will need to create the credentials for connecting to the service to configure the bridge from Message Hub.

Select the Service Credentials tab and click the New Credential button to add the new credentials.



Confirm adding the new credentials by clicking Add.

Now click the View Credentials button for the key `Credentials-1`.



Click the  button to copy the credentials into the clipboard.

Start the Text Editor application from the launcher on the left-hand side of your screen.

In the editor, press Ctrl+V to paste the credentials from the clipboard. It will look something like this:

Creating Object Storage Bridges in Message Hub

Thankfully, there is now a GUI to configure the bridge. Please follow the "Corrected Instructions" document to configure the bridge

Now, you need to create an Object Storage Bridge. This will automatically take all of the messages arriving on the Message Hub `carevents` topic and copy the data into Object Storage for historical analysis. The container in Object Storage is also called `carevents`.

Open a terminal window, and make sure you're in the /home/demo directory.

Create the following environment variables needed to configure the bridge like this:

```
export OS_AUTH_URL='your auth_url in the editor'
export OS_REGION='your region value in the editor'
export OS_PASSWORD='your password in the editor'
export OS_PROJECT_ID='your projectId in the editor'
export OS_USER_ID='your userId in the editor'
```

It's a really good idea to use single quotes like this because authentication tokens and passwords sometimes have peculiar characters in them.

You'll also need the API key for connecting to Message Hub which you can find in the Message Hub service credentials list. Create an environment variable for this too:

```
export MH_API_KEY='your Message Hub API key'
```

Then run the following script to create the bridge:

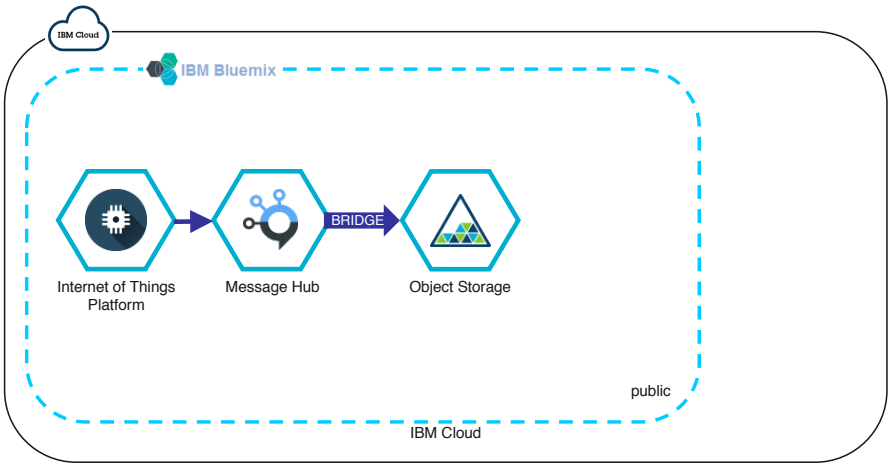
```
./createObjectStorageBridge
```

You can use the following script to check the status:

```
./listBridges
```

Recap

It's worth taking a moment to look at what you've just created.



You have a Message Hub instance with a topic called `carevents`, IoT Platform configured to send the events from all devices to this topic, and an Object Storage Bridge transferring the events and storing them in Object Storage.

Part 2 – Generating a stream of events to analyze

Imagine that you have a small fleet of connected cars. Every now and again, each car connects to the cloud and sends in a snapshot of its state. Here's an example of the data:

```
{
  "vehicleNumber" : "car18",
  "fuelRemaining" : "5.28",
  "fuelCapacity" : "11.0",
  "distance" : "3866.690",
  "refills" : "1",
  "ts" : "2017-02-02T22:26:59.110"
}
```

This is an event from `car18`. It has 5.28 gallons of fuel remaining and a total capacity of 11.0 gallons. It has travelled 3866.690 miles. Since the last event, it has refilled once.

The lab uses a small Java application called `SimulatedCar`. The application simulates 20 cars. Every second, it chooses a car at random and instructs it to send an event. The application calculates the car's speed and how fast it's consuming fuel.

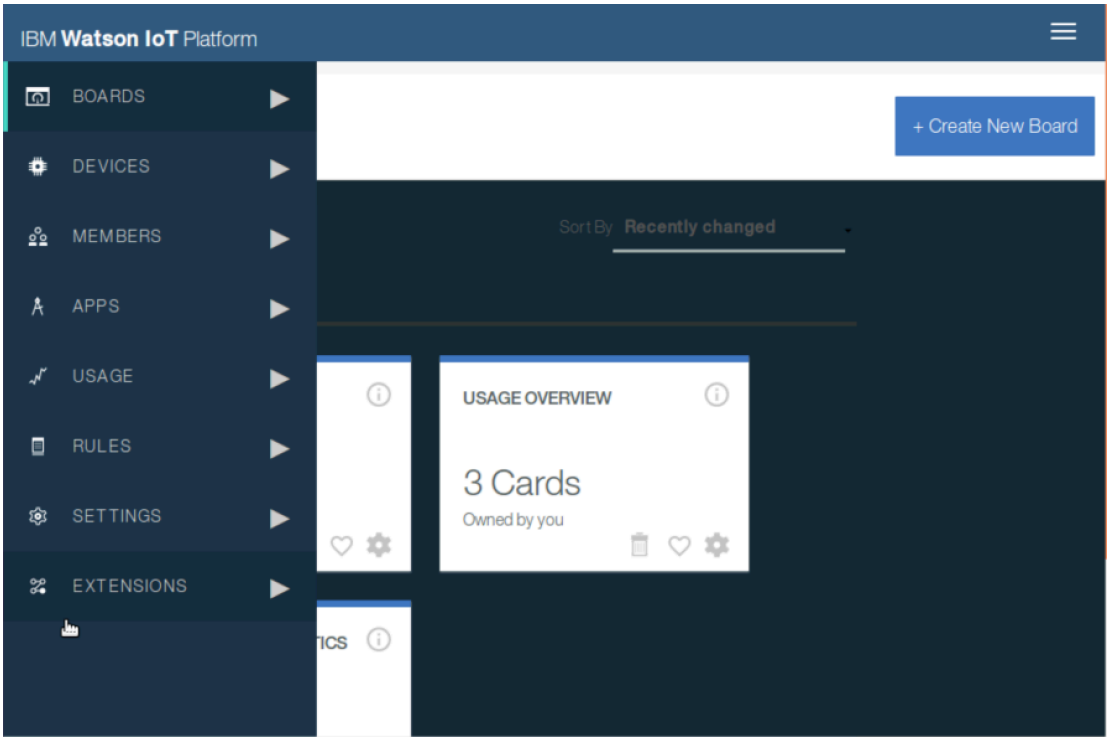
Although an event is sent every second, the timestamp between consecutive events is actually 15 minutes. This means that once the application has run for a few minutes, it will produce events with timestamps spanning multiple days.

Registering devices in IoT Platform for the simulated cars

Each simulated car needs to be registered in IoT Platform as a device which generates a set of credentials the simulated car uses to connect and send in the events.

You could do this using the IoT Platform dashboard, but it's a bit laborious adding 20 cars. In a real deployment, you'd have an automated process to register the cars as they were manufactured. You'll use an application to perform this role so first you need to register this application with IoT Platform.

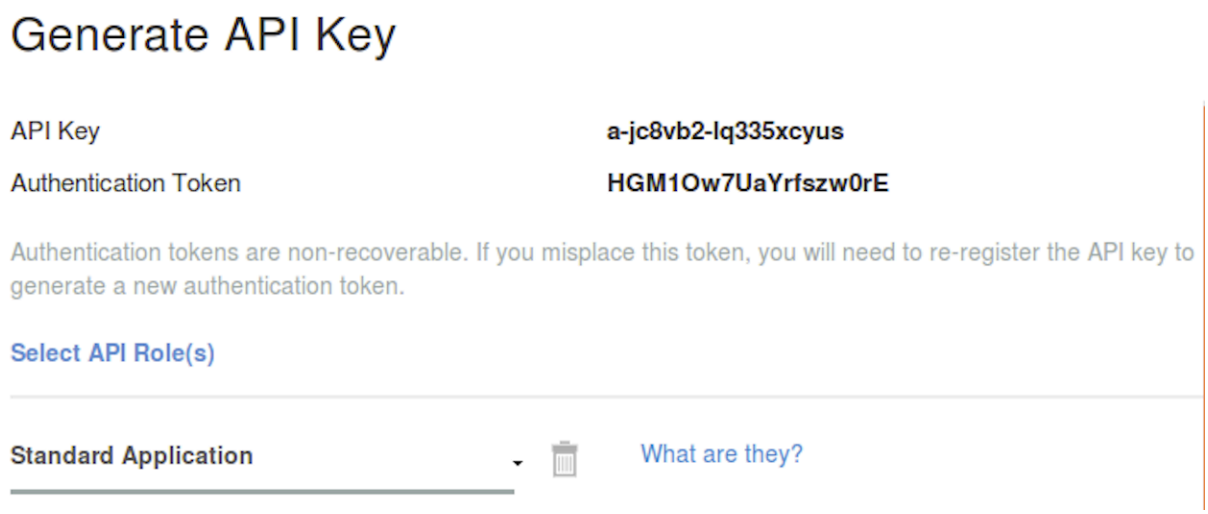
In the web browser, go to the IoT Platform dashboard.



Remember that you can get here from the IoT Platform service in the Bluemix console by pressing:

Launch

In the menu, select APPS. Then, click the Generate API key button to register get credentials for your application:



Of course, your API key and authentication token will be different. You need to capture these values since you'll need them to register your devices.

Open a terminal window, and make sure you're in the /home/demo directory.

Create two environment variables for the API key and authentication token like this:

```
export IOT_API_KEY='your API key'
export IOT_AUTH_TOKEN='your authentication token'
```

Then, confirm the generation of the API key by pressing the Generate button. If you don't do this, the API key is displayed and then discarded.

when you copy and paste these lines, please remove the space behind export IOT_ORG_ID

You'll also need the organization ID which you can find in the IoT Platform dashboard, either in the top-right corner or the first 6 characters of the URL for the dashboard. Create an environment variable for this too:

```
export IOT_ORG_ID ='your org_id'
```

Then, run the following script to create the definitions in IoT Platform for the simulated cars:

```
./createIoTDevices
```

Please replace the path in the runSimulatedCars script with the one where it is stored at your environment

If you look in the IoT Platform dashboard, you'll see 20 devices defined to represent the cars.

Generating events from the simulated cars

Now you can run the simulated cars application by typing:

```
./runSimulatedCars
```

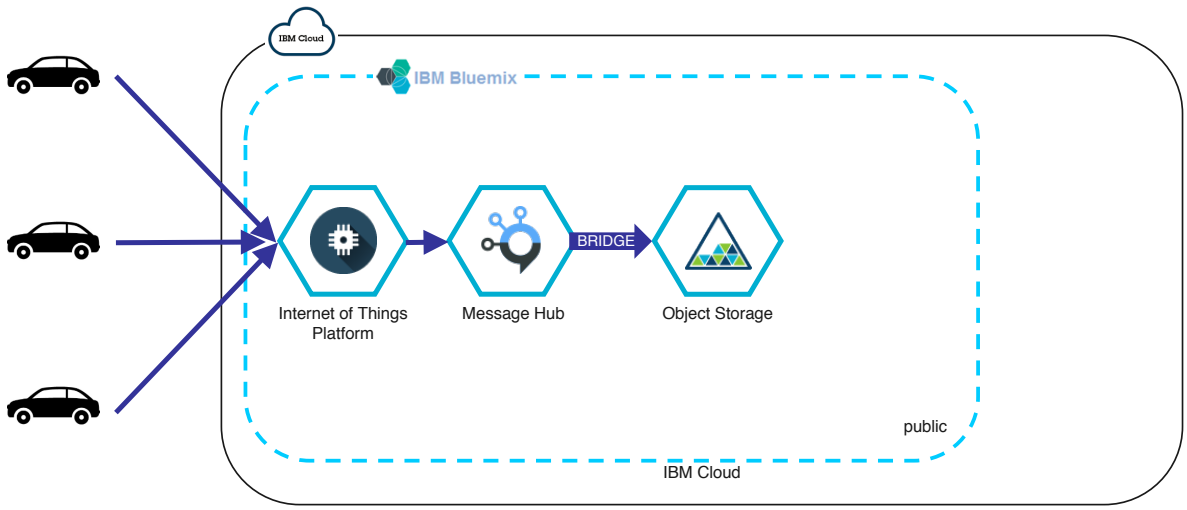
This connects to IoT Platform as a device, or more accurately as a device for each of the cars sending an event. The application prints out the information in each event just before it is sent to IoT Platform.

It's a simple Java application and you can find the source on the lab laptop.

You should leave this application running for the rest of the lab so that it continues to produce a stream of events to process.

Recap

Here's what you've created so far.



The events from the cars will be visible in the IoT Platform dashboard if you go to the table of devices, select a device and wait a moment for that car to send an event. After a few minutes, you will also be able to see the events arrive in Object Storage.

The next step is the analysis.

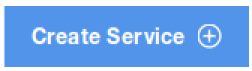
Part 3 – Analysis of data in Object Storage

Using the Data Science Experience service, we're able to perform analytics in a web browser in an interactive environment called a notebook. This section of the lab uses a Jupyter notebook and the Spark SQL module to analyze events stored in Object Storage.

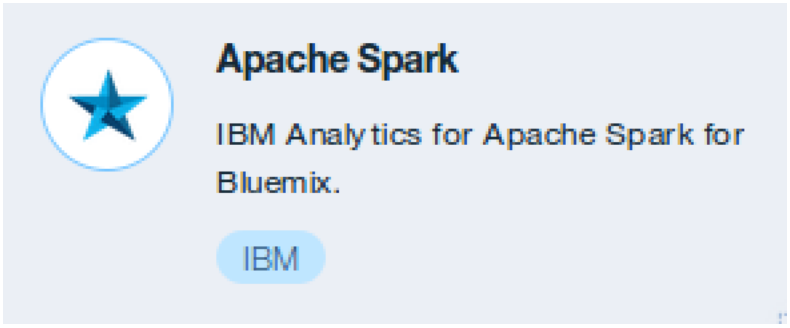
You're going to need two more services: Apache Spark to run the analytics and Data Science Experience to provide the user interface.

Creating an Apache Spark instance

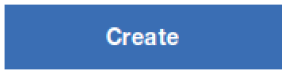
Go back to the services dashboard in the Bluemix console and begin the process of creating another service instance by clicking this button:



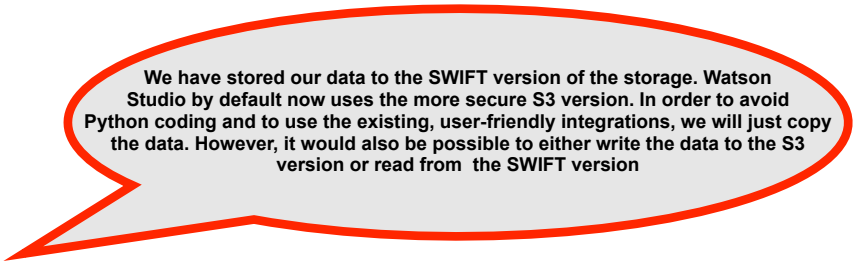
In the services catalog that you can now see, find the Apache Spark tile and click it:

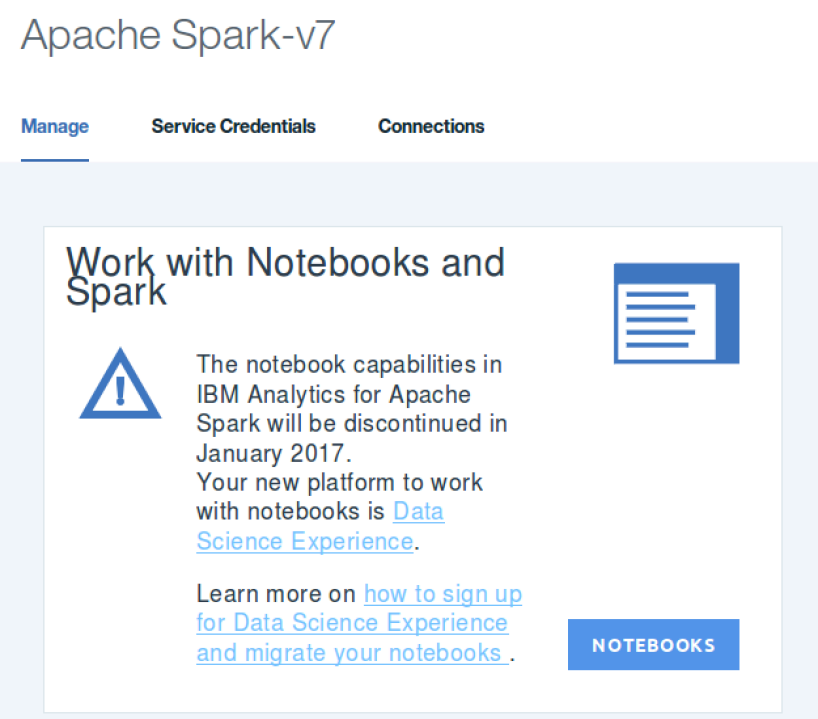


The next page describes the service, how to use it and how much it costs. Click the Create button to create your service instance:

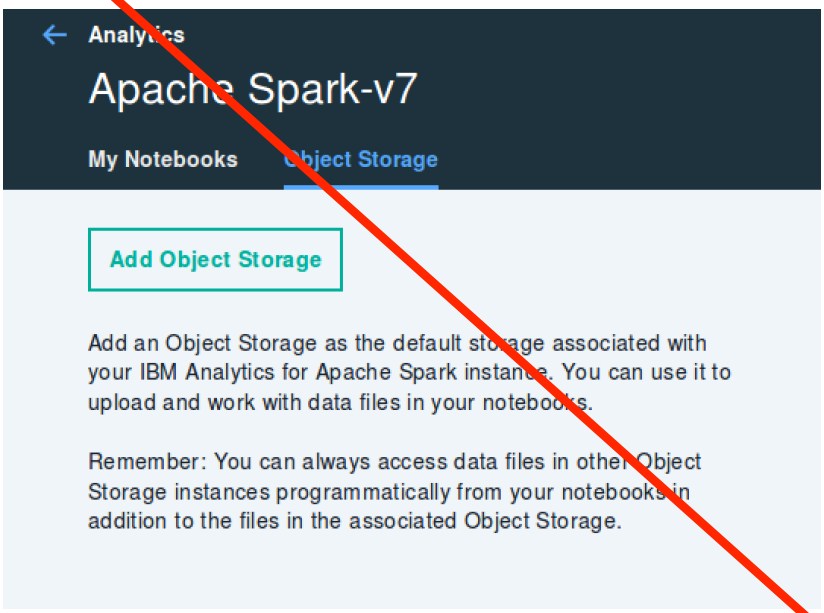


After a few seconds, you'll go into the Apache Spark service UI.





You need to point Apache Spark at the Object Storage service instance you created earlier. Click the NOTEBOOKS button to manage notebooks and then select the Object Storage tab. The screen now looks like this:



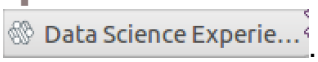
Click Add Object Storage and then select the Bluemix tab.

Note that this step will only succeed once the Object Storage Bridge has begun to transfer data. This is because it creates the container into which it places the data. If you've been working really quickly, you might need to pause now. Alternatively, you can go back, create an empty container in Object Storage called `carevents`, and then return to this step.

The instance of Object Storage service that you created is already selected. Click SELECT to confirm.

Now you'll be able to refer to containers in Object Storage directly in your analytics code.

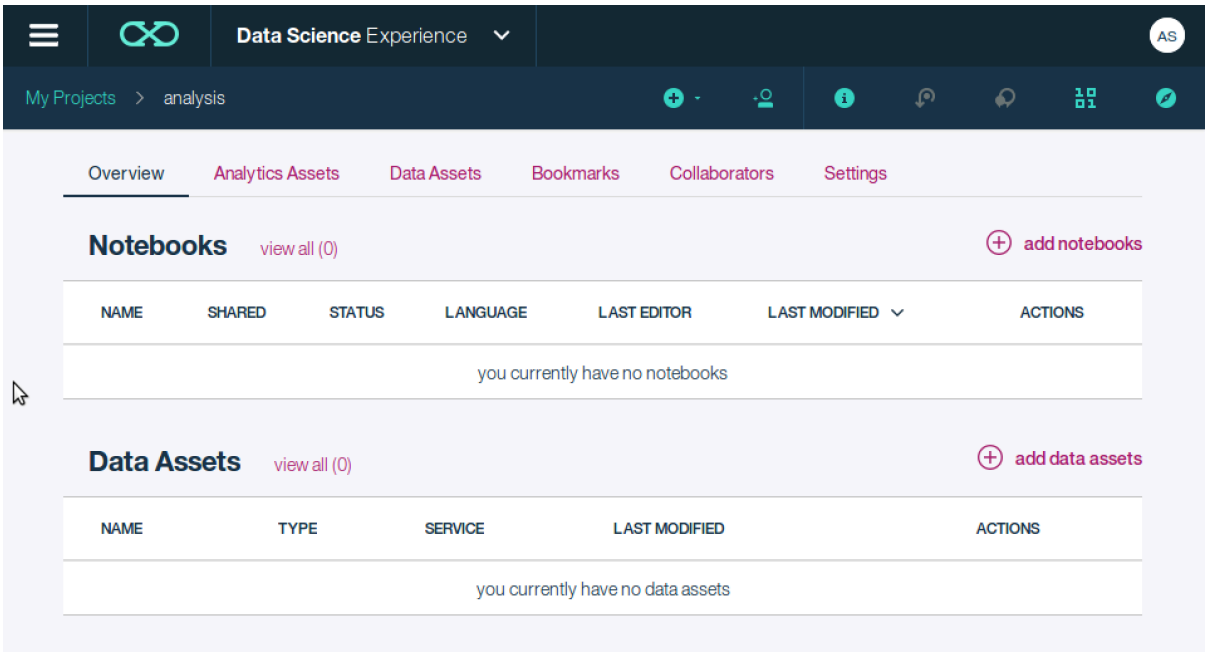
Analysis using Data Science Experience

In the bookmarks toolbar in Firefox, click 

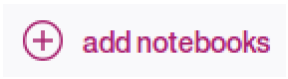
.

Create a new project and call it `analysis`. The instance of Object Storage will be pre-selected and a new container of the same name will be created automatically.

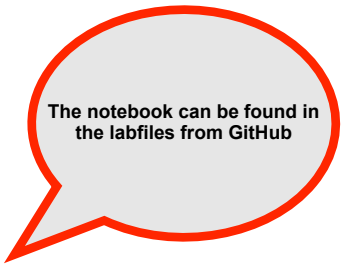
Once the project has been created, it will look like this:

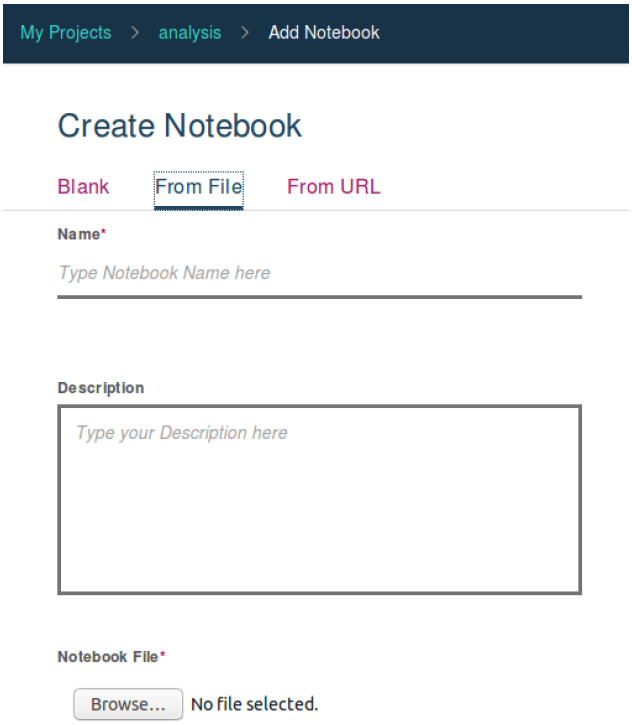


Now, create a notebook for the analysis logic by clicking:



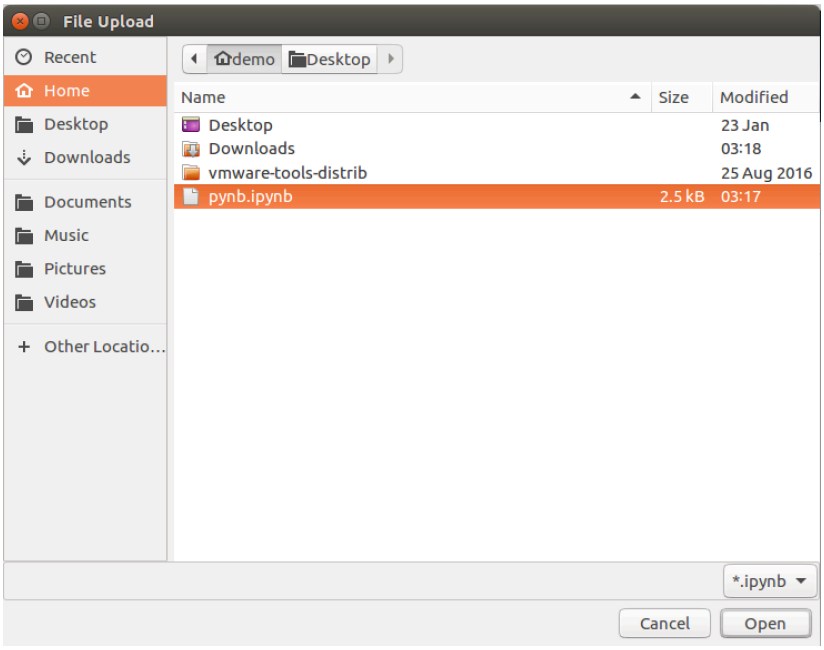
The notebook is in the `/home/demo` directory on the laptop, so choose the From File tab.





Specify pynb as the name of the notebook.

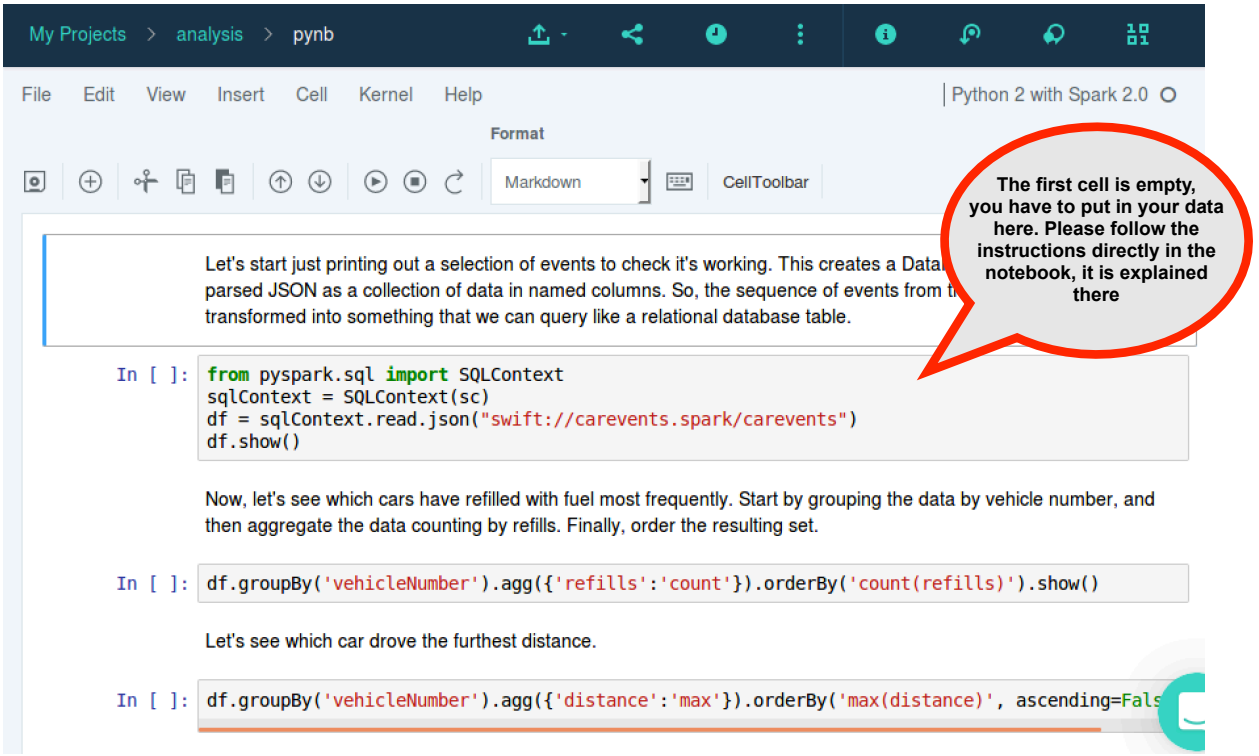
Press the Browse… button to select a notebook file.



Choose the file `pynb.ipynb` from the Home directory and confirm by pressing Open.

Click Create Notebook to complete the creation of the notebook.

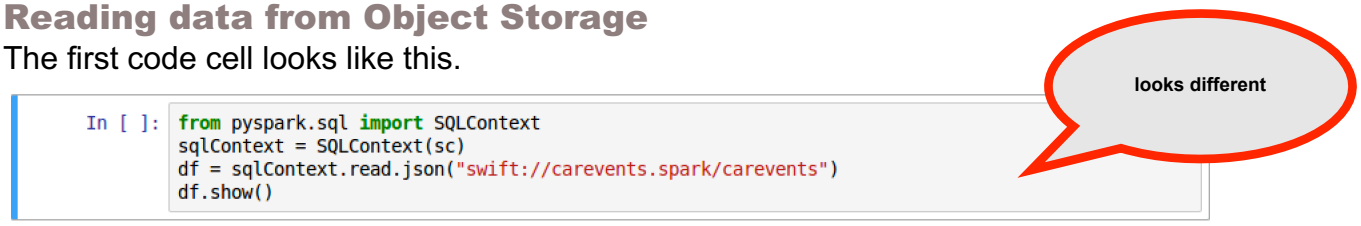
After a few moments, an Apache Spark kernel starts up and you enter the notebook editor.



The notebook consists of a list of cells, some of which are description in markdown format and some are code. This notebook uses Python code.

Reading data from Object Storage

The first code cell looks like this.



To run the code, click on the cell to select it and press  . After a short delay, the output is displayed beneath the code.

```
+------------------+------------+--------------------+-------+--------------------+-------------+----------+
|          distance|fuelCapacity|       fuelRemaining|refills|                  ts|vehicleNumber|        dt|
+------------------+------------+--------------------+-------+--------------------+-------------+----------+
| 291.5841360226474|        11.0|0.018950078403076276|      0|2017-02-03T00:12:...|        car11|2017-02-03|
| 834.5159548331014|        11.0|  3.2236155825590593|      1|2017-02-03T01:12:...|        car16|2017-02-03|
| 605.3932257989513|        11.0|   9.564149131147795|      1|2017-02-03T00:27:...|        car17|2017-02-03|
| 713.5929415822962|        11.0|   6.682158258261055|      1|2017-02-03T00:42:...|        car18|2017-02-03|
| 771.7861883550087|        11.0|   4.330868954001565|      1|2017-02-03T00:57:...|         car3|2017-02-03|
| 936.1125956200619|        11.0|    3.95184368940507|      0|2017-02-03T01:27:...|         car5|2017-02-03|
| 971.1247774705479|        11.0|   5.332603424935375|      1|2017-02-03T02:12:...|        car10|2017-02-03|
| 864.4615913648161|        11.0|  1.9458281230947367|      0|2017-02-03T01:42:...|         car9|2017-02-03|
| 882.1985801344935|        11.0|  1.3459842298689078|      0|2017-02-03T01:57:...|         car9|2017-02-03|
| 827.3815258539819|        11.0|   1.652046447811177|      1|2017-02-03T02:42:...|         car7|2017-02-03|
| 983.3442194230681|        11.0|   6.275909717928712|      1|2017-02-03T02:27:...|         car2|2017-02-03|
| 872.8136732570484|        11.0|  1.3474487254191834|      0|2017-02-03T02:57:...|        car18|2017-02-03|
| 866.7603872523105|        11.0| 0.18721702846575505|      0|2017-02-03T03:12:...|         car7|2017-02-03|
| 879.8225117335654|        11.0|  10.781230561388941|      1|2017-02-03T03:27:...|         car7|2017-02-03|
| 831.7360890366718|        11.0|   5.301793229734045|      2|2017-02-03T03:42:...|        car19|2017-02-03|
```

```
| 954.7856204979397|       11.0|   6.752602754593873|      3|2017-02-03T04:57:...|        car4|2017-02-03|
| 971.1247774705479|       11.0|   5.332603424935375|      0|2017-02-03T03:57:...|       car10|2017-02-03|
|1393.3510211938508|       11.0|   9.718987104398877|      5|2017-02-03T04:12:...|       car12|2017-02-03|
| 796.2101169884476|       11.0|  6.8800045148980224|      2|2017-02-03T04:27:...|        car1|2017-02-03|
| 879.8225117335654|       11.0|  10.781230561388941|      0|2017-02-03T04:42:...|        car7|2017-02-03|
+------------------+-----------+--------------------+-------+--------------------+------------+----------+
only showing top 20 rows
```

So, what's it doing? This just prints the first 20 rows of data it reads from the `carevents` container in Object Storage. Because you associated the Apache Spark service with the Object Storage service, you can just attach the SQL context to `swift://`*`container`*`.spark/`*`object`* to read the data. Also, Spark SQL is parsing the JSON events, inferring the schema of the data and converting the events into tuples. Each line of the input file must contain a separate, self-contained JSON object, and that's exactly what the Object Storage Bridge generates.

You can find out more by looking at the Spark SQL documentation here: https://spark.apache.org/docs/2.0.1/api/python/pyspark.sql.html.

More complex queries

Let's say that we want to find out which cars have refilled most often. To do this, we want to group the events by `vehicleNumber`, count the number of refills in the groups and order the results.

```
df.groupBy('vehicleNumber').agg({'refills':'count'}).orderBy('count(
refills)').show()
```

You could use SQL SELECT syntax instead.

The notebook you loaded contains this query to so try running it.

Similarly, you can see which car has driven furthest like this.

```
df.groupBy('vehicleNumber').agg({'distance':'max'}).orderBy('max(dis
tance)', ascending=False).take(1)
```

To try your own experiment, add another cell to the notebook, type in your code and run it. You can find out more from the Spark SQL documentation here: https://spark.apache.org/docs/2.0.1/api/python/pyspark.sql.html.

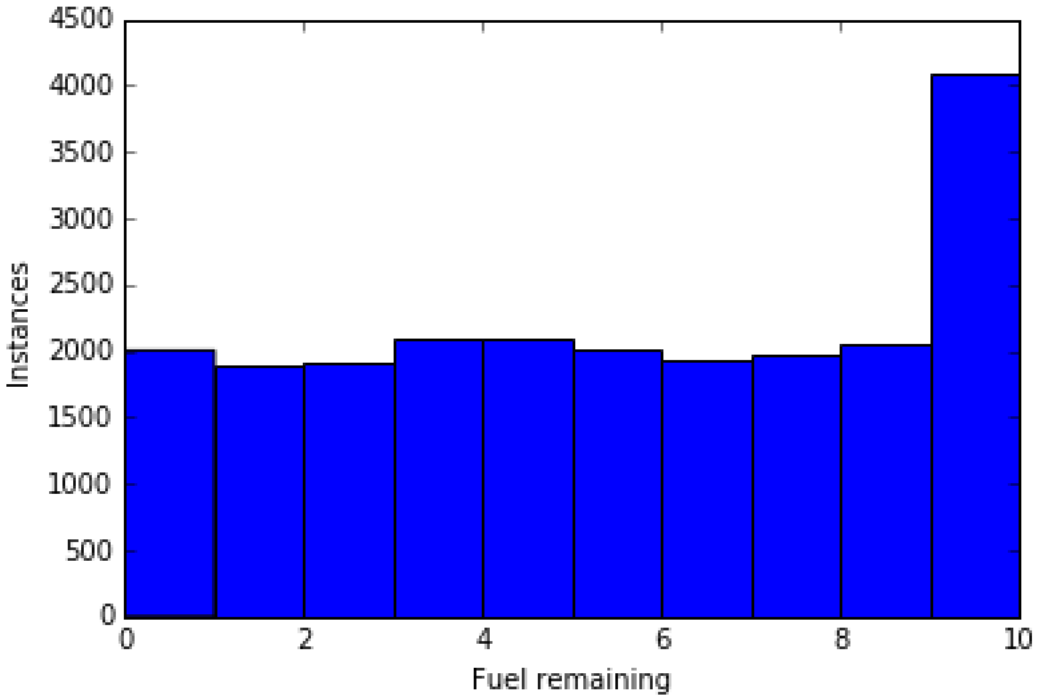
Visualization in the browser

You can also plot the data directly in the browser. For example, this code draws a histogram of the distribution of `fuelRemaining` amounts.

```
%matplotlib inline
import matplotlib.pyplot as plt
df.registerTempTable('carevents')
xx = []
d = sqlContext.sql("select fuelRemaining from carevents").collect()
for row in d:
    xx.append(int(getattr(row, "fuelRemaining")))
plt.hist(xx)
plt.xlabel('Fuel remaining')
plt.ylabel('Instances')
plt.show()
```

You can see that this example is using SQL syntax (`select fuelRemaining from carevents`).

If you run this cell, you should see output something like this:



In this case, my simulator has been running for several days and there are thousands of events to process. But the result was still calculated very quickly so it's practical to experiment with the queries in an interactive way.

The matplotlib plotting library is very powerful and you can find out much more by reading the documentation here: http://matplotlib.org.

Part 4 – Streaming analytics

You can use Apache Spark to analyze events right out of Message Hub. This extension to the Spark API that enables stream processing of live data streams. You can find more information here: http://spark.apache.org/docs/latest/streaming-programming-guide.html.

Spark Streaming takes the live data from an input stream, divides it into batches which are then processed. The stream of events is known as an input DStream. The lab shows how you can manipulate an input DStream to transform it and extract the data you want to analyze.

Although the processing operates in batches, you can also maintain state across batches and perform analyses across time windows. The lab shows how to calculate a running total using a Spark Streaming accumulator.

Preparing to connect to Message Hub

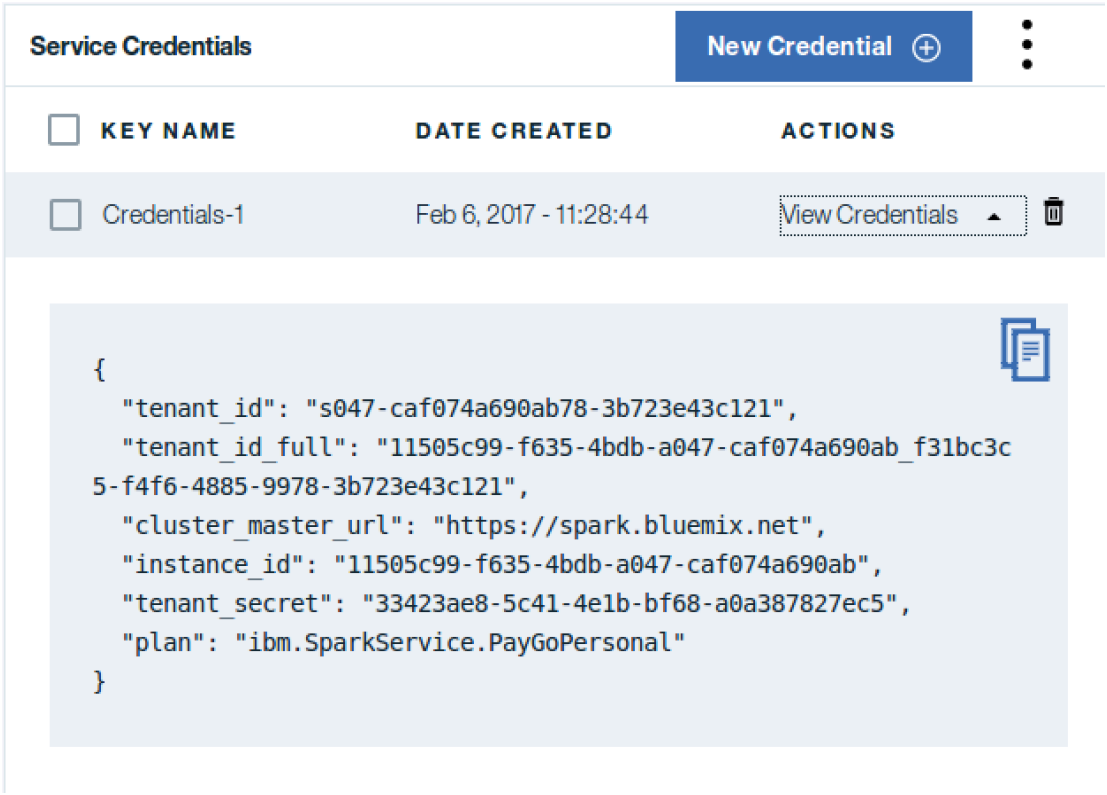
The core of the Message Hub service is an open-source project called Apache Kafka. As a result, the Apache Spark Streaming support for Kafka works with Message Hub. So, you'll be using a Kafka client to connect to Message Hub.

While it's possible to get the standard Spark Streaming support for Kafka to work with Message Hub, it takes a bit of work to get the credentials mechanism in the Kafka 0.10 client to work properly in Spark.

The Kafka 0.10.2 client adds an enhancement which makes it much simpler to handle the credentials. In time, Spark Streaming will bundle this client, but for the lab, it's simplest just to upload the Kafka 0.10.2 client JAR into your Apache Spark environment.

Uploading the Kafka 0.10.2 client JAR

In the Bluemix console, navigate to your Apache Spark service instance. Select the Service Credentials tab and click the View Credentials button for the key `Credentials-1`.



Open a terminal window, and make sure you're in the /home/demo directory.

Create three environment variables for the Spark user ID, password and instance ID like this:

```
export SPARK_TENANT_ID='your Spark tenant_id'
export SPARK_TENANT_SECRET='your Spark tenant_secret'
export SPARK_INSTANCE_ID='your Spark instance_id'
```

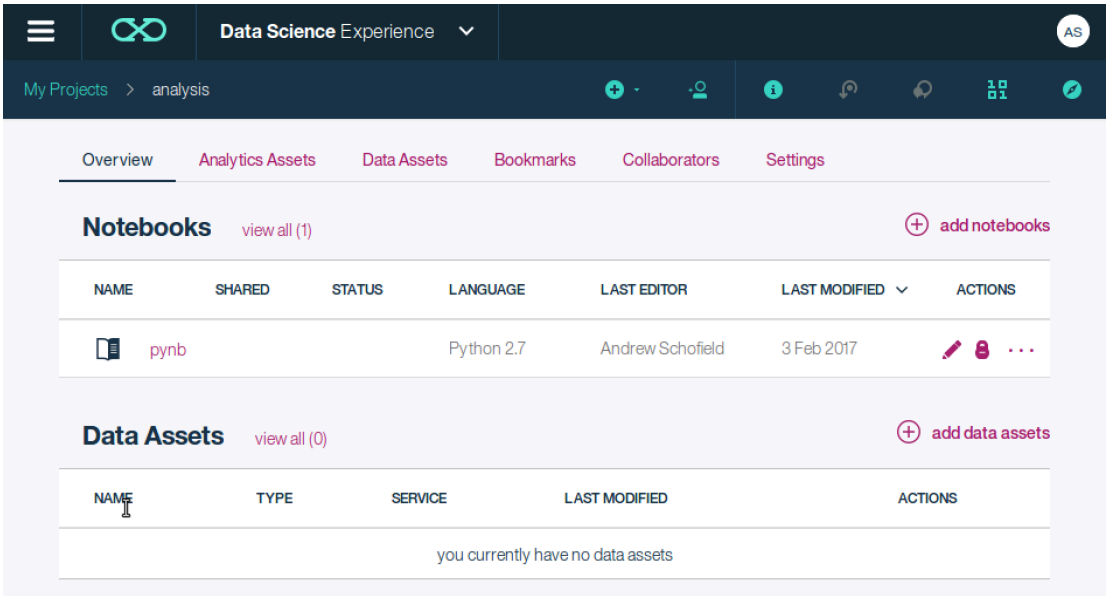
Then, run the following script to upload the Kafka 0.10.2 clients JAR:

```
./uploadKafkaJar
```

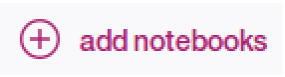
The JAR file is copied into the `data/libs` directory of your Apache Spark instance which inserts it into the classpath ahead of the built-in Kafka client.

Checking the classpath in Spark

In Firefox, navigate to the analysis project you created in Data Science Experience. It will look like this:



Now, create a notebook by clicking:



Specify `checkupload` as the name of the notebook, select Python 2 as the language and Spark 2.0 as the Spark Version.

Click Create Notebook to complete the creation of the notebook.

After a few moments, an Apache Spark kernel starts up and you enter the notebook editor. Enter the following code into the empty code cell:

```
%%bash
cd
ls data/libs
```

To run the code, click on the call to select it and press  . After a short delay, the output is displayed beneath the code.

Check the output which should look like this:

```
kafka-clients-0.10.2.0.jar
```

This shows that the JAR file was uploaded into the right place.

Streaming data into Apache Spark

Add another notebook from a file on the laptop. This time, specify `streaming-print` as the name of the notebook, choose the file `streaming-print.ipynb` from the Home directory and create the notebook.

This notebook is a bit more complicated and this time it's written in Scala. Let's look at the code.

The first line tells Spark that we want to use the Kafka support for Spark Streaming. It downloads the appropriate JAR files and installs them.

```
%AddDeps org.apache.spark spark-streaming-kafka-0-10_2.11 2.0.2
```

After the imports section, this code sets the Kafka parameters.

```
val kafkaParams = Map[String, Object](
  "bootstrap.servers" -> "kafka01-prod01.messagehub.services.us-
south.bluemix.net:9093",
  "key.deserializer" -> classOf[StringDeserializer],
  "value.deserializer" -> classOf[StringDeserializer],
  "group.id" -> "streaminganalysis",
  "security.protocol" -> "SASL_SSL",
  "sasl.mechanism" -> "PLAIN",
  "ssl.protocol" -> "TLSv1.2",
  "ssl.enabled.protocols" -> "TLSv1.2",
  "ssl.endpoint.identification.algorithm" -> "HTTPS",
  "sasl.jaas.config" ->
"org.apache.kafka.common.security.plain.PlainLoginModule required
username=\"$MH_USER_ID$\" password=\"$MH_PASSWORD$\";",
  "auto.offset.reset" -> "latest",
  "enable.auto.commit" -> (false: java.lang.Boolean)
)
```

The line that starts `sasl.jaas.config` contains the credentials for connecting to Message Hub. Support for this configuration property is the enhancement that we want to use from the Kafka 0.10.2 client. You need to replace `$MH_USER_ID$` and `$MH_PASSWORD$` with the credentials from your Message Hub instance.

The next line gives the name of the topic we want to read for events.

```
val topics = Array("carevents")
```

Then let's skip the next bit and look at the code that actually sets up the Spark Streaming environment.

```
def creatingFunc(sc: SparkContext): StreamingContext = {
  val ssc = new StreamingContext(sc, Seconds(5))

  val stream = KafkaUtils.createDirectStream[String, String](
    ssc,
    PreferConsistent,
    Subscribe[String, String](topics, kafkaParams)
  )
```

Because of the way that Spark Streaming is initialized, it's best to us a creating function like this. The idea is that, when the code runs, it first checks to see if Spark Streaming has been initialized. If it has not, it runs the creating function.

The function creates a `StreamingContext` and sets the batch interval to 5 seconds. That means that every 5 seconds, the events that have been received are processed in a batch.

Then the code creates a direct stream by connecting to Message Hub and subscribing to the `carevents` topic. The variable stream is a stream of Kafka `ConsumerRecord` elements.

The rest of the function maps the data from the incoming `ConsumerRecord` format, to strings containing JSON documents and finally into a stream of name-value pairs. The code could use a JSON parser, but it's a relatively straightforward sequence of manipulations and you can see better how to work with Spark Streaming.

This line has the effect of printing out the data in its final form. So, we expect every 5 seconds the output to show the events that arrived in the latest batch.

```
eventsAsMaps.print
```

The final lines of the code perform the initialization checks, start the processing and wait forever.

```
val sc = SparkContext.getOrCreate()
val ssc = StreamingContext.getActiveOrCreate(() => creatingFunc(sc))
ssc.start()
ssc.awaitTermination()
```

Once you've changed the credentials, start the code in the usual way by pressing  .

After a brief pause, you should see output being generated. Initially, the batches will not contain any records because it takes a while for Message Hub to assign partitions to this consumer. But soon you should see data like this being printed out every 5 seconds.

```
-------------------------------------------
Time: 1486392130000 ms
-------------------------------------------
Map(vehicleNumber -> car1, fuelRemaining -> 10.06, fuelCapacity ->
11.0, refills -> 2, distance -> 724.92)
```

This isn't really analysis yet, so let's try something more interesting.

Streaming analysis in Apache Spark

Apache Spark is a very powerful environment for data processing. It's capable of distributing the workload across a cluster of executing nodes. When used with Kafka, the partitions of a topic can be assigned to different executors so that the massive scalability of Kafka can be matched by scaling the analytics code.

The way that Spark Streaming has been designed makes all of this easy. By importing the Kafka 0.10.2 JAR into the project and passing the credentials when creating the direct stream, Message Hub fits nicely into the framework.

Of course, when the workload is being distributed, it makes it more difficult to aggregate results. Luckily, Spark provides features to help here.

The analysis we're going to perform is simply to calculate a running total of the number of refills of the cars. To do that, we're going to use an accumulator.

Add another notebook from a file on the laptop. This time, specify `streaming-accumulator` as the name of the notebook, choose the file `streaming-accumulator.ipynb` from the Home directory and create the notebook.

This notebook is exactly the same as the previous one, with the addition of a running total using an accumulator.

```
object RefillsAccumulator {
  @volatile private var instance: LongAccumulator = null

  def getInstance(sc: SparkContext): LongAccumulator = {
    if (instance == null) {
      synchronized {
        if (instance == null) {
          instance = sc.longAccumulator("RefillsAccumulator")
        }
      }
    }
    instance
  }
}
```

This code creates an accumulator called `RefillsAccumulator`. Most the code is ensuring that a single instance is created safely.

The code to calculate the running total looks like this:

```
val refills = eventsAsMaps.map(_("refills").toLong)

// Accumulate the number of refills over time
refills.foreachRDD{ (rdd: RDD[Long], time: Time) =>
  val refillsAccumulator = RefillsAccumulator.getInstance(rdd.s...
  rdd.foreach{r => refillsAccumulator.add(r)}
  println(\"Refills: \" + refillsAccumulator.value);
}
```

This takes the maps we printed earlier, and just takes the refills values. Then for each value, it adds to the accumulator.

Once you've changed the credentials, start the code in the usual way by pressing  .

When you run the code, the output should be an increasing number reflecting the refilling of the cars in the simulation.

```
Refills: 0
Refills: 1
Refills: 6
```

What next?

This is a very simple example of streaming analysis, but it gives a flavor of what's possible. Spark Streaming has a wealth of features for more complex analysis beyond the scope of this lab.

Why not try some more complex charting using matplotlib, or analysis using a time window?