Let's have some fun with code

In previous introduction notebook we showed you, how simple is it to use Kafka by yourself.

In order to stream PID data, we used something very similar. Downloading a data, parse them and send to broker server, which is being hosted on AWS. So basicaly we producing data and now you have to consume them.

We will use pyspark and spark structured streaming API.

It is not the only one option, how to consume kafka topics. Kafka itself has APIs to manage it fully, see more here.

First, we have to connect to Kafka's broker. The broker is hosted as MSK Kafka AWS Service.

1. *broker1 b-1-public.bdffelkafka.3jtrac.c19.kafka.us-east-1.amazonaws.com:9196*
2. *broker2 b-2-public.bdffelkafka.3jtrac.c19.kafka.us-east-1.amazonaws.com:9196*

You will need user name and password, provided on previous lectures.

We have 5 topics from which we can read: trams, buses, regbuses, trains and boats, we will start with trams for now.

```
# connect to broker
JAAS = 'org.apache.kafka.common.security.scram.ScramLoginModule required username="USERXX" password="PWDXX";'
tram_stream_topic = spark.readStream \
  .format("kafka")\
  .option("kafka.bootstrap.servers", "b-2-public.bdffelkafka.3jtrac.c19.kafka.us-east-1.amazonaws.com:9196," \
  "b-1-public.bdffelkafka.3jtrac.c19.kafka.us-east-1.amazonaws.com:9196") \
  .option("kafka.sasl.mechanism", "SCRAM-SHA-512")\
  .option("kafka.security.protocol", "SASL_SSL") \
  .option("kafka.sasl.jaas.config", JAAS) \
  .option("subscribe", "trams") \
  .load()
```

It is possible to cast json messages directly, using schema, or you can cast it just to string, but it might complicate your work later.

Use schema below. It is possible to save it to other python or dbx notebook and call it externally, see example below for dbx notebook.

```
%run "./pid_schema" # the notebook's name with function in it
schema_pid = get_pid_schema() # use it for casting later
```

To call the schema, it must be wrapped into function with return.

The schema is uploaded on github, copy paste it to separate notebook, or right away in the same notebook.

```

from pyspark.sql.functions import from_json, col
base_trams = tram_stream_topic.select(from_json(col("value").cast("string"), schema_pid).alias("data")).select("data.*") \
```

Let's start the actual spark stream. There are several ways, how to store data, which output mode - what type of sink we will use, documentation is here.

Now we use format `memory` - data will be stored in memory and we will append, ie. we do not wait for complete data (in specified batch or so).

Memory sink is nice for debugging, but you have to be really sure, that data are not big too much. Data are being saved in memory table on Spark's driver.

Supported mode is `Append` or `Complete`.

```
#
tram_stream_mem_append = base_trams.writeStream \
         .format("memory")\
         .queryName("mem_trams")\
         .outputMode("append")\
         .start()
```

Using files is useful, when you'd like to append data and save bigger amount and process it later.

You can pull hourly data from stream to your file and analyze it after few days. This can be done by running trigger option and scheduling notebook to run every hour in Workflows menu.

```
tram_stream_file = base_trams.writeStream \
        .format("parquet")\
        .option("path", "path/to/destination/dir")
        .start()
#.trigger(once=True).format("delta").queryName(query_name).outputMode("append")
# .option("checkpointLocation", "/Filestore/whatever/").toTable("nameoftable")
```

We can try now some basic SQL operations on the mem_trams table.

```
%sql
select * from mem_trams;
```

We can check how many entries do we have.

```
%sql
select count(*) from mem_trams;
```

Just remember, if you want to make some transformations on the data, you should not do it directly on stream, since you could corrupt the data in it.

```
%sql
create table data_trams select * from mem_trams;
```

Now we can do some transformation on the data_trams. For example we can print coordinations of the first 10 trams.

```
%sql
select geometry.coordinates from data_trams limit 10;

```

Ok, let's divide an array with coords.

```
%sql
SELECT
  cast(data_trams.geometry.coordinates[0] as double) AS x,
  cast(data_trams.geometry.coordinates[1] as double) AS y
FROM data_trams;
```

To visualize geopoints, you can use geopy, matplotlib, openstreetmaps and osmnx libs... we will be using some of them. First we need to install libraries.

```
%pip install osmnx
```

```
%pip install numpy==1.23.0
```

Import oxmns and create custom filter (more links you have, longer you will wait). You can create graph by center, by polygon etc. See [OSMNX docs]()

```
import osmnx as ox

custom_filter='["highway"~"motorway|motorway_link|trunk|trunk_link|primary|primary_link|secondary|secondary_link|road|road_link"]'

G = ox.graph_from_place("Praha, Czechia", custom_filter=custom_filter)
```

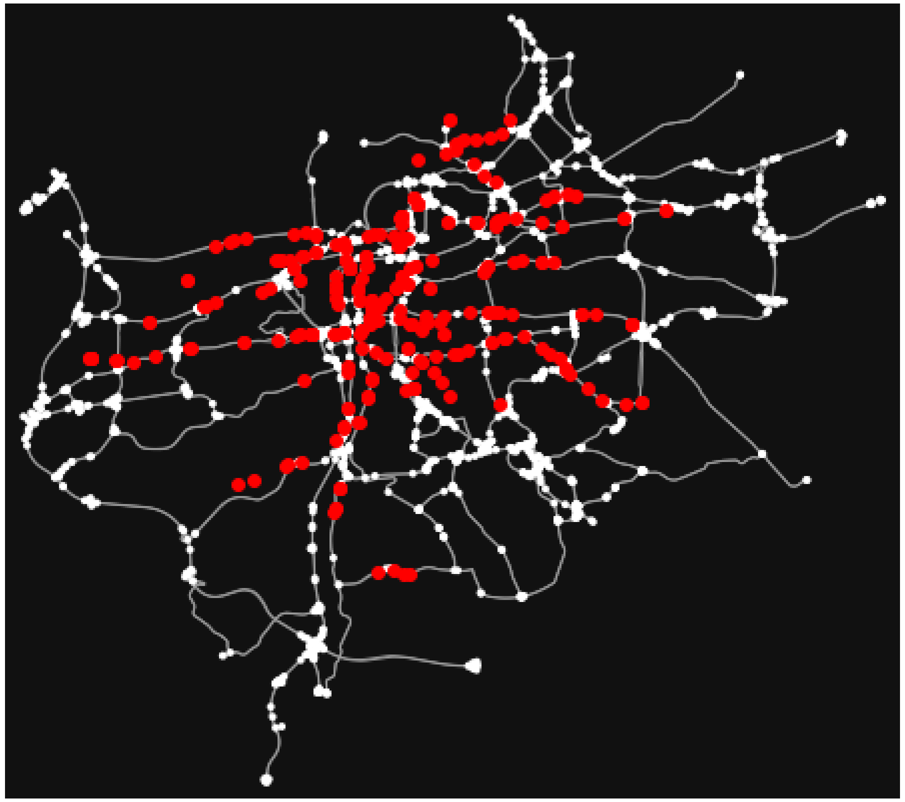
Once you graph is loaded, you can plot it out together with your x and y coordintaes (obtainted via select Spark above).

```
import matplotlib.pyplot as plt
# this makes your plot wait and not closing
fig, ax = ox.plot_graph(G, show=False, close=False)
df_geo_p = df_geo.toPandas()
# you can plot all, or some subsection for quicker result
x = df_geo_p.loc[1:300,'x']
y = df_geo_p.loc[1:300,'y']

ax.scatter(x, y, c='red')

plt.show()
```

The results is the map below.



Have fun with geo spatial data :)